跟我学 Spring 系列



目錄

介紹 0

跟我学 Spring3 1

【第二章】 IoC 之 2.1 IoC基础 ——跟我学Spring3 1.1

【第二章】 IoC 之 2.2 IoC 容器基本原理 ——跟我学Spring3 1.2

【第二章】 IoC 之 2.3 IoC的配置使用——跟我学Spring3 1.3

【第三章】 DI 之 3.1 DI的配置使用 ——跟我学spring3 1.4

【第三章】 DI 之 3.2 循环依赖 ——跟我学spring3 1.5

【第三章】 DI 之 3.3 更多DI的知识 ——跟我学spring3 1.6

【第三章】 DI 之 3.4 Bean的作用域 ——跟我学spring3 1.7

【第四章】 资源 之 4.1 基础知识 ——跟我学spring3 1.8

【第四章】 资源 之 4.2 内置Resource实现 ——跟我学spring3 1.9

【第四章】 资源 之 4.3 访问Resource ——跟我学spring3 1.10

【第四章】 资源 之 4.4 Resource通配符路径 ——跟我学spring3 1.11

【第五章】Spring表达式语言 之 5.1 概述 5.2 SpEL基础 ——跟我学spring3 1.12

【第五章】Spring表达式语言 之 5.3 SpEL语法 ——跟我学spring3 1.13

【第五章】Spring表达式语言 之 5.4在Bean定义中使用EL—跟我学spring3 1.14

【第六章】 AOP 之 6.1 AOP基础 ——跟我学spring3 1.15

【第六章】 AOP 之 6.2 AOP的HelloWorld ——跟我学spring3 1.16

【第六章】 AOP 之 6.3 基于Schema的AOP ——跟我学spring3 1.17

【第六章】 AOP 之 6.4 基于@AspectJ的AOP ——跟我学spring3 1.18

【第六章】 AOP 之 6.5 AspectJ切入点语法详解 ——跟我学spring3 1.19

【第六章】 AOP 之 6.6 通知参数 ——跟我学spring3 1.20

【第六章】 AOP 之 6.7 通知顺序 ——跟我学spring3 1.21

【第六章】 AOP 之 6.8 切面实例化模型 ——跟我学spring3 1.22

【第六章】 AOP 之 6.9 代理机制 ——跟我学spring3 1.23

【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3 1.24

【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3 1.25

【第七章】 对JDBC的支持 之 7.3 关系数据库操作对象化 ——跟我学spring3 1.26

【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3【私塾在线原创】 1.27

【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

【第八章】 对ORM的支持 之 8.1 概述 ——跟我学spring3 1.29 1.28
【第八章】 对ORM的支持 之 8.2 集成Hibernate3 ——跟我学spring3 1.30
【第八章】 对ORM的支持 之 8.3 集成iBATIS ——跟我学spring3 1.31
【第八章】 对ORM的支持 之 8.4 集成JPA ——跟我学spring3 1.32
【第九章】 Spring的事务 之 9.1 数据库事务概述 ——跟我学spring3 1.33
【第九章】 Spring的事务 之 9.2 事务管理器 ——跟我学spring3 1.34
【第九章】 Spring的事务 之 9.3 编程式事务 ——跟我学spring3 1.35
【第九章】 Spring的事务 之 9.4 声明式事务 ——跟我学spring3 1.36
【第十章】集成其它Web框架 之 10.1 概述 ——跟我学spring3 1.37
【第十章】集成其它Web框架 之 10.2 集成Struts1.x ——跟我学spring3 1.38
【第十章】集成其它Web框架 之 10.3 集成Struts2.x ——跟我学spring3 1.39
【第十章】集成其它Web框架 之 10.4 集成JSF ——跟我学spring3 1.40
【第十一章】 SSH集成开发积分商城 之 11.1 概述 ——跟我学spring3 1.41
【第十一章】 SSH集成开发积分商城 之 11.2 实现通用层 ——跟我学spring3 1.42
【第十一章】 SSH集成开发积分商城 之 11.3 实现积分商城层 ——跟我学spring3
【第十二章】零配置 之 12.1 概述 ——跟我学spring3 1.44 1.43
【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3 1.45
【第十二章】零配置 之 12.3 注解实现Bean定义 ——跟我学spring3 1.46
【第十二章】零配置 之 12.4 基于Java类定义Bean配置元数据 ——跟我学spring3
【第十二章】零配置 之 12.5 综合示例-积分商城 ——跟我学spring3 1.48 1.47
【第十三章】 测试 之 13.1 概述 13.2 单元测试 ——跟我学spring3 1.49
【第十三章】 测试 之 13.3 集成测试 ——跟我学spring3 1.50
跟我学 Spring MVC 2
SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结 2.1
Spring Web MVC中的页面缓存支持 ——跟我学SpringMVC系列 2.2
Spring3 Web MVC下的数据类型转换（第一篇）——《跟我学Spring3 Web MVC》抢先看 2.3
Spring3 Web MVC下的数据格式化（第二篇）——《跟我学Spring3 Web MVC》抢先看 2.4
第一章 Web MVC简介 —— 跟开涛学SpringMVC 2.5
第二章 Spring MVC入门 —— 跟开涛学SpringMVC 2.6
第三章 DispatcherServlet详解 ——跟开涛学SpringMVC 2.7
第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC 2.8
第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC 2.9

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC 2.10
第四章 Controller接口控制器详解 （4）——跟着开涛学SpringMVC 2.11
第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC 2.12
跟着开涛学SpringMVC 第一章源代码下载 2.13
第二章 Spring MVC入门 源代码下载 2.14
第四章 Controller接口控制器详解 源代码下载 2.15
第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC 2.16
第四章 Controller接口控制器详解（7 完）——跟着开涛学SpringMVC 2.17
第五章 处理器拦截器详解——跟着开涛学SpringMVC 2.18
源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC 2.19
注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC 2.20
源代码下载 第六章 注解式控制器详解 2.21
SpringMVC3强大的请求映射规则详解 第六章 注解式控制器详解——跟着开涛学SpringMVC 2.22
Spring MVC 3.1新特性 生产者、消费者请求限定 —— 第六章 注解式控制器详解——跟着开涛学SpringMVC 2.23
SpringMVC强大的数据绑定（1）——第六章 注解式控制器详解——跟着开涛学SpringMVC 2.24
SpringMVC强大的数据绑定（2）——第六章 注解式控制器详解——跟着开涛学SpringMVC 2.25
SpringMVC数据类型转换——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC 2.26
SpringMVC数据格式化——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC 2.27
SpringMVC数据验证——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC 2.28

跟我学 Spring 系列

跟我学 Spring3

> 作者：开涛
>
> 来源：跟我学spring3

【第二章】 IoC 之 2.1 IoC基础 ——跟我学Spring3

2.1.1 IoC是什么

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好Ioc呢？理解好Ioc的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●谁控制谁，控制什么：传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建；谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

用图例说明一下，传统程序设计如图2-1，都是主动去创建相关对象然后再组合起来：

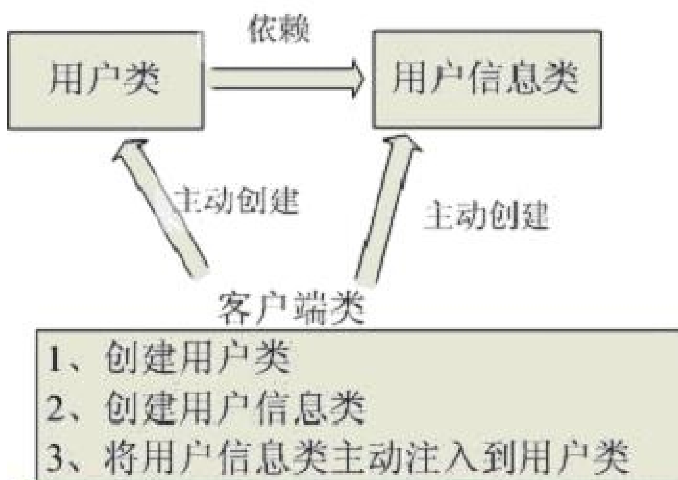


图2-1 传统应用程序示意图

当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图2-2所示:

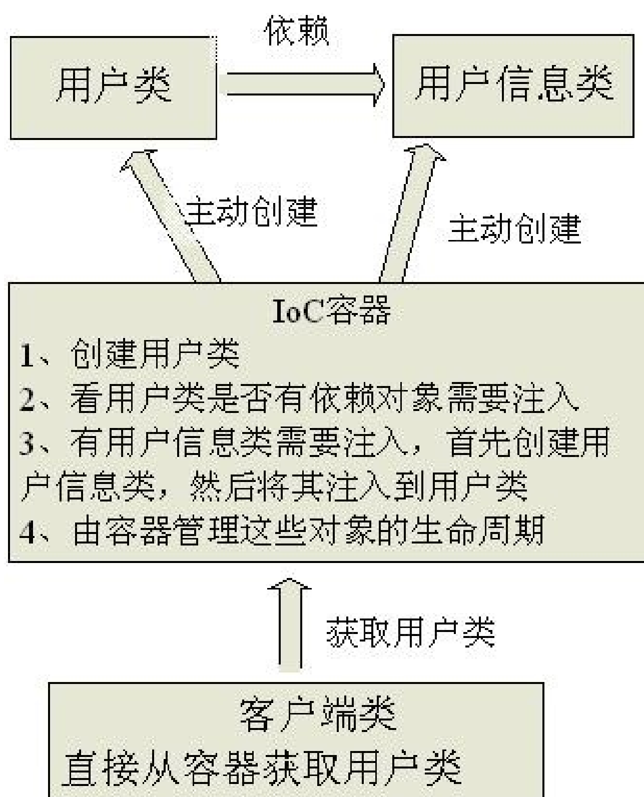


图2-2有IoC/DI容器后程序结构示意图

1.1.2 IoC能做什么

IoC不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了"主从换位"的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。

IoC很好的体现了面向对象设计法则之一—— 好莱坞法则："别找我们，我们找你"；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

2.1.3 IoC和DI

DI—Dependency Injection，即"依赖注入"：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是："谁依赖谁，为什么需要依赖，谁注入谁，注入了什么"，那我们来深入分析一下：

●谁依赖于谁：当然是应用程序依赖于IoC容器；

●为什么需要依赖：应用程序需要IoC容器来提供对象需要的外部资源；

●谁注入谁：很明显是IoC容器注入应用程序某个对象，应用程序依赖的对象；

●注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC和DI由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字："依赖注入"，相对IoC 而言，**"依赖注入"明确描述了"被注入对象依赖IoC容器配置依赖对象"**。

注：如果想要更加深入的了解IoC和DI，请参考大师级人物Martin Fowler的一篇经典文章《Inversion of Control Containers and the Dependency Injection pattern》，原文地址：http://www.martinfowler.com/articles/injection.html。

转自【http://sishuok.com/forum/blogPost/list/2427.html】

【第二章】IoC 之 2.2 IoC 容器基本原理 ——跟我学Spring3

2.2.1 IoC容器的概念

IoC容器就是具有依赖注入功能的容器，IoC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中new相关的对象，应用程序由IoC容器进行组装。在Spring中BeanFactory是IoC容器的实际代表者。

Spring IoC容器如何知道哪些是它管理的对象呢？这就需要配置文件，Spring IoC容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于xml配置文件进行配置元数据，而且Spring与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于java文件的、基于属性文件的配置都可以。

那Spring IoC容器管理的对象叫什么呢？

2.2.2 Bean的概念

由IoC容器管理的那些组成你应用程序的对象我们就叫它Bean， Bean就是由Spring容器初始化、装配及管理的对象，除此之外，bean就与应用程序中的其他对象没有什么区别了。那IoC怎样确定如何实例化Bean、管理Bean之间的依赖关系以及管理Bean呢？这就需要配置元数据，在Spring中由BeanDefinition代表，后边会详细介绍，配置元数据指定如何实例化Bean、如何组装Bean等。概念知道的差不多了，让我们来做个简单的例子。

2.2.3 Hello World

一、配置环境：

l **JDK**安装：安装最新的JDK，至少需要Java 1.5及以上环境；

l 开发工具：SpringSource Tool Suite，简称STS，是个基于Eclipse的开发环境，用以构建Spring应用，其最新版开始支持Spring 3.0及OSGi开发工具，但由于其太庞大，很多功能不是我们所必需的所以我们选择Eclipse+ SpringSource Tool插件进行Spring应用开发；到eclipse官网下载最新的Eclipse，注意我们使用的是Eclipse IDE for Java EE Developers（eclipse-jee-helios-SR1）；

安装插件：启动Eclipse，选择Help->Install New Software，如图2-3所示





图2-3 安装

2、首先安装SpringSource Tool Suite插件依赖，如图2-4:

Name为：SpringSource Tool Suite Dependencies

Location为：http://dist.springsource.com/release/TOOLS/composite/e3.6



图2-4 安装

3、安装SpringSource Tool Suite插件，只需安装如图2-5所选中的就可以：

Name为：SpringSource Tool Suite

Location为：http://dist.springsource.com/release/TOOLS/update/e3.6



图2-4 安装

4、安装完毕，开始项目搭建吧。

Spring 依赖：本书使用spring-framework-3.0.5.RELEASE

spring-framework-3.0.5.RELEASE-with-docs.zip表示此压缩包带有文档的；

spring-framework-3.0.5.RELEASE-dependencies.zip表示此压缩包中是spring的依赖jar包，所以需要什么依赖从这里找就好了；

下载地址：http://www.springsource.org/download

二、开始Spring Hello World之旅

1、准备需要的jar包

核心**jar**包：从下载的spring-framework-3.0.5.RELEASE-with-docs.zip中dist目录查找如下jar包

```
org.springframework.asm-3.0.5.RELEASE.jar
org.springframework.core-3.0.5.RELEASE.jar
org.springframework.beans-3.0.5.RELEASE.jar
org.springframework.context-3.0.5.RELEASE.jar
org.springframework.expression-3.0.5.RELEASE.jar
```

依赖的**jar**包：从下载的spring-framework-3.0.5.RELEASE-dependencies.zip中查找如下依赖jar包

```
com.springsource.org.apache.log4j-1.2.15.jar
com.springsource.org.apache.commons.logging-1.1.1.jar
com.springsource.org.apache.commons.collections-3.2.1.jar
```

2、创建标准Java工程：

（1）选择“window”—> “Show View” —>“Package Explorer”，使用包结构视图；



图2-5 包结构视图

（2）创建标准Java项目，选择“File”—>“New”—>“Other”；然后在弹出来的对话框中选择“Java Project”创建标准Java项目；



图2-6 创建Java项目



图2-7 创建Java项目



图2-8 创建Java项目

（3）配置项目依赖库文件，右击项目选择“Properties”；然后在弹出的对话框中点击“Add JARS”在弹出的对话框中选择“lib”目录下的jar包；然后再点击“Add Library”，然后在弹出的对话框中选择“Junit”，选择“Junit4”；



图2-9 配置项目依赖库文件



图2-10 配置项目依赖库文件





图2-11 配置项目依赖库文件

（4）项目目录结构如下图所示，其中“src”用于存放java文件；“lib”用于存放jar文件；“resources”用于存放配置文件；



图2-12 项目目录结构

3、项目搭建好了，让我们来开发接口，此处我们只需实现打印“Hello World!”，所以我们定义一个“sayHello”接口，代码如下：

```
1.  package cn.javass.spring.chapter2.helloworld;
2.  public interface HelloApi {
3.  public void sayHello();
4.  }
```

4、接口开发好了，让我们来通过实现接口来完成打印“Hello World!”功能；

```
1.  package cn.javass.spring.chapter2.helloworld;
2.  public class HelloImpl implements HelloApi {
3.  @Override
4.  public void sayHello() {
5.  System.out.println("Hello World!");
6.  }
7.  }
```

5、接口和实现都开发好了，那如何使用Spring IoC容器来管理它们呢？这就需要配置文件，让IoC容器知道要管理哪些对象。让我们来看下配置文件chapter2/helloworld.xml（放到resources目录下）：

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans
3.  xmlns="http://www.springframework.org/schema/beans"
4.  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.  xmlns:context="http://www.springframework.org/schema/context"
6.  xsi:schemaLocation="
7.  http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8.  http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd"&gt;
9.  <!-- id 表示你这个组件的名字，class表示组件类 -->
10. <bean id="hello" class="cn.javass.spring.chapter2.helloworld.HelloImpl"></bean>
11. </beans>
```

6、现在万一具备，那如何获取IoC容器并完成我们需要的功能呢？首先应该实例化一个IoC容器，然后从容器中获取需要的对象，然后调用接口完成我们需要的功能，代码示例如下：

```
1.  package cn.javass.spring.chapter2.helloworld;
2.  import org.junit.Test;
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.  public class HelloTest {
```

```
6. @Test
7. public void testHelloWorld() {
8. //1、读取配置文件实例化一个IoC容器
9. ApplicationContext context = new ClassPathXmlApplicationContext("helloworld.xml");
10. //2、从容器中获取Bean，注意此处完全“面向接口编程，而不是面向实现”
11. HelloApi helloApi = context.getBean("hello", HelloApi.class);
12. //3、执行业务逻辑
13. helloApi.sayHello();
14. }
15. }
```

7、自此一个完整的Spring Hello World已完成，是不是很简单，让我们深入理解下容器和Bean吧。

2.2.4 详解IoC容器

在Spring Ioc容器的代表就是org.springframework.beans包中的BeanFactory接口，BeanFactory接口提供了IoC容器最基本功能；而org.springframework.context包下的ApplicationContext接口扩展了BeanFactory，还提供了与Spring AOP集成、国际化处理、事件传播及提供不同层次的context实现 (如针对web应用的WebApplicationContext)。简单说，BeanFactory提供了IoC容器最基本功能，而 ApplicationContext 则增加了更多支持企业级功能支持。ApplicationContext完全继承BeanFactory，因而BeanFactory所具有的语义也适用于ApplicationContext。

容器实现一览：

• **XmlBeanFactory**：BeanFactory实现，提供基本的IoC容器功能，可以从classpath或文件系统等获取资源；

（1） File file = new File("fileSystemConfig.xml");

Resource resource = new FileSystemResource(file);

BeanFactory beanFactory = new XmlBeanFactory(resource);

（2）

Resource resource = new ClassPathResource("classpath.xml");

BeanFactory beanFactory = new XmlBeanFactory(resource);

• **ClassPathXmlApplicationContext**：ApplicationContext实现，从classpath获取配置文件；

BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath.xml");

• **FileSystemXmlApplicationContext**：ApplicationContext实现，从文件系统获取配置文件。

BeanFactory beanFactory = new FileSystemXmlApplicationContext("fileSystemConfig.xml");

具体代码请参考cn.javass.spring.chapter2.InstantiatingContainerTest.java。

ApplicationContext接口获取Bean方法简介：

• Object getBean(String name) 根据名称返回一个Bean，客户端需要自己进行类型转换；

• T getBean(String name, Class<T> requiredType) 根据名称和指定的类型返回一个Bean，客户端无需自己进行类型转换，如果类型转换失败，容器抛出异常；

• T getBean(Class<T> requiredType) 根据指定的类型返回一个Bean，客户端无需自己进行类型转换，如果没有或有多于一个Bean存在容器将抛出异常；

• Map<String, T> getBeansOfType(Class<T> type) 根据指定的类型返回一个键值为名字和值为Bean对象的 Map，如果没有Bean对象存在则返回空的Map。

让我们来看下IoC容器到底是如何工作。在此我们以xml配置方式来分析一下：

一、准备配置文件：就像前边Hello World配置文件一样，在配置文件中声明Bean定义也就是为Bean配置元数据。

二、由**IoC**容器进行解析元数据： IoC容器的Bean Reader读取并解析配置文件，根据定义生成BeanDefinition配置元数据对象，IoC容器根据BeanDefinition进行实例化、配置及组装Bean。

三、实例化**IoC**容器：由客户端实例化容器，获取需要的Bean。

整个过程是不是很简单，执行过程如图2-5，其实IoC容器很容易使用，主要是如何进行Bean定义。下一章我们详细介绍定义Bean。



图2-5 Spring Ioc容器

2.2.5 小结

除了测试程序的代码外，也就是程序入口，所有代码都没有出现Spring任何组件，而且所有我们写的代码没有实现框架拥有的接口，因而能非常容易的替换掉Spring，是不是非入侵。

客户端代码完全面向接口编程，无需知道实现类，可以通过修改配置文件来更换接口实现，客户端代码不需要任何修改。是不是低耦合。

如果在开发初期没有真正的实现，我们可以模拟一个实现来测试，不耦合代码，是不是很方便测试。

Bean之间几乎没有依赖关系，是不是很容易重用。

转自【http://sishuok.com/forum/blogPost/list/2428.html】

【第二章】 IoC 之 2.3 IoC的配置使用——跟我学Spring3

2.3.1 XML配置的结构

一般配置文件结构如下：

```
1. <beans>
2. <import resource=”resource1.xml”/>
3. <bean id=”bean1”class=””></bean>
4. <bean id=”bean2”class=””></bean>
5. <bean name=”bean2”class=””></bean>
6. <alias alias="bean3" name="bean2"/>
7. <import resource=”resource2.xml”/>
8. </beans>
```

1、<bean>标签主要用来进行Bean定义；

2、alias用于定义Bean别名的；

3、import用于导入其他配置文件的Bean定义，这是为了加载多个配置文件，当然也可以把这些配置文件构造为一个数组（new String[] {“config1.xml”, config2.xml}）传给ApplicationContext实现进行加载多个配置文件，那一个更适合由用户决定；这两种方式都是通过调用Bean Definition Reader 读取Bean定义，内部实现没有任何区别。<import>标签可以放在<beans>下的任何位置，没有顺序关系。

2.3.2 Bean的配置

Spring IoC容器目的就是管理Bean，这些Bean将根据配置文件中的Bean定义进行创建，而Bean定义在容器内部由BeanDefinition对象表示，该定义主要包含以下信息：

●全限定类名（FQN）：用于定义Bean的实现类;

●Bean行为定义：这些定义了Bean在容器中的行为；包括作用域（单例、原型创建）、是否惰性初始化及生命周期等；

●Bean创建方式定义：说明是通过构造器还是工厂方法创建Bean；

●Bean之间关系定义：即对其他bean的引用，也就是依赖关系定义，这些引用bean也可以称之为同事bean 或依赖bean，也就是依赖注入。

Bean定义只有“全限定类名”在当使用构造器或静态工厂方法进行实例化bean时是必须的，其他都是可选的定义。难道Spring只能通过配置方式来创建Bean吗？回答当然不是，某些SingletonBeanRegistry接口实现类实现也允许将那些非BeanFactory创建的、已有的用户对象注册到容器中，这些对象必须是共享的，比如使用DefaultListableBeanFactory 的registerSingleton() 方法。不过建议采用元数据定义。

2.3.3 Bean的命名

每个Bean可以有一个或多个id（或称之为标识符或名字），在这里我们把第一个**id**称为**“**标识符**”**，其余**id**叫做**“**别名**”**；这些id在IoC容器中必须唯一。如何为Bean指定id呢，有以下几种方式；

一、 不指定id，只配置必须的全限定类名，由IoC容器为其生成一个标识，客户端必须通过接口“T getBean(Class<T> requiredType)”获取Bean；

```
1. <bean class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/> （1）
```

测试代码片段如下：

```
1. @Test
2. public void test1() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean1.xml");
5. //根据类型获取bean
6. HelloApi helloApi = beanFactory.getBean(HelloApi.class);
7. helloApi.sayHello();
8. }
```

二、指定id，必须在Ioc容器中唯一；

```
1. <bean id=” bean” class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/> （2）
```

测试代码片段如下：

```
1. @Test
2. public void test2() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean2.xml");
5. //根据id获取bean
6. HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
7. bean.sayHello();
8. }
```

三、指定name，这样name就是“标识符”，必须在Ioc容器中唯一；

```
1. <bean name=” bean” class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/> （3）
```

测试代码片段如下：

```
1. @Test
2. public void test3() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean3.xml");
5. //根据name获取bean
6. HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
7. bean.sayHello();
8. }
```

四、指定id和name，id就是标识符，而name就是别名，必须在Ioc容器中唯一；

```
1. <bean id=”bean1”name=”alias1”
2. class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/>
3. <!-- 如果id和name一样，IoC容器能检测到，并消除冲突 -->
4. <bean id="bean3" name="bean3"
   class="cn.javass.spring.chapter2.helloworld.HelloImpl"/> （4）
```

测试代码片段如下：

```
1. @Test
2. public void test4() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean4.xml");
5. //根据id获取bean
6. HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
7. bean1.sayHello();
8. //根据别名获取bean
9. HelloApi bean2 = beanFactory.getBean("alias1", HelloApi.class);
10. bean2.sayHello();
11. //根据id获取bean
12. HelloApi bean3 = beanFactory.getBean("bean3", HelloApi.class);
13. bean3.sayHello();
14. String[] bean3Alias = beanFactory.getAliases("bean3");
15. //因此别名不能和id一样，如果一样则由IoC容器负责消除冲突
16. Assert.assertEquals(0, bean3Alias.length);
17. }
```

五、指定多个name，多个name用“，”、“；”、“ ”分割，第一个被用作标识符，其他的（alias1、alias2、alias3）是别名，所有标识符也必须在Ioc容器中唯一；

```
1. <bean name=” bean1;alias11,alias12;alias13 alias14”
2. class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/>
3. <!-- 当指定id时，name指定的标识符全部为别名 -->
4. <bean id="bean2" name="alias21;alias22"
5. class="cn.javass.spring.chapter2.helloworld.HelloImpl"/> （5）
```

测试代码片段如下：

```
1. @Test
2. public void test5() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean5.xml");
5. //1根据id获取bean
6. HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
7. bean1.sayHello();
8. //2根据别名获取bean
9. HelloApi alias11 = beanFactory.getBean("alias11", HelloApi.class);
10. alias11.sayHello();
11. //3验证确实是四个别名
12. String[] bean1Alias = beanFactory.getAliases("bean1");
13. System.out.println("=======namingbean5.xml bean1 别名========");
14. for(String alias : bean1Alias) {
15. System.out.println(alias);
16. }
17. Assert.assertEquals(4, bean1Alias.length);
18. //根据id获取bean
19. HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
20. bean2.sayHello();
21. //2根据别名获取bean
22. HelloApi alias21 = beanFactory.getBean("alias21", HelloApi.class);
23. alias21.sayHello();
24. //验证确实是两个别名
25. String[] bean2Alias = beanFactory.getAliases("bean2");
26. System.out.println("=======namingbean5.xml bean2 别名========");
27. for(String alias : bean2Alias) {
28. System.out.println(alias);
29. }
30. Assert.assertEquals(2, bean2Alias.length);
31. }
```

六、使用<alias>标签指定别名，别名也必须在IoC容器中唯一

```
1. <bean name="bean" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <alias alias="alias1" name="bean"/>
3. <alias alias="alias2" name="bean"/> （6）
```

测试代码片段如下：

```
1. @Test
2. public void test6() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean6.xml");
5. //根据id获取bean
6. HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
7. bean.sayHello();
8. //根据别名获取bean
9. HelloApi alias1 = beanFactory.getBean("alias1", HelloApi.class);
10. alias1.sayHello();
11. HelloApi alias2 = beanFactory.getBean("alias2", HelloApi.class);
12. alias2.sayHello();
13. String[] beanAlias = beanFactory.getAliases("bean");
14. System.out.println("=======namingbean6.xml bean 别名========");
15. for(String alias : beanAlias) {
16. System.out.println(alias);
17. }
18. System.out.println("=======namingbean6.xml bean 别名========");
19. Assert.assertEquals(2, beanAlias.length);
20. }
```

以上测试代码在cn.javass.spring.chapter2.NamingBeanTest.java文件中。

从定义来看，name或id如果指定它们中的一个时都作为"标识符"，那为什么还要有id和name同时存在呢？这是因为当使用基于XML的配置元数据时，在XML中id是一个真正的XML id属性，因此当其他的定义来引用这个id时就体现出id的好处了，可以利用XML解析器来验证引用的这个id是否存在，从而更早的发现是否引用了一个不存在的bean，而使用name，则可能要在真正使用bean时才能发现引用一个不存在的bean。

●**Bean**命名约定：Bean的命名遵循XML命名规范，但最好符合Java命名规范，由"字母、数字、下划线组成"，而且应该养成一个良好的命名习惯， 比如采用"驼峰式"，即第一个单词首字母开始，从第二个单词开始首字母大写开始，这样可以增加可读性。

2.3.4 实例化Bean

Spring IoC容器如何实例化Bean呢？传统应用程序可以通过new和反射方式进行实例化Bean。而Spring IoC容器则需要根据Bean定义里的配置元数据使用反射机制来创建Bean。在Spring IoC容器中根据Bean定义创建Bean主要有以下几种方式：

一、使用构造器实例化Bean：这是最简单的方式，Spring IoC容器即能使用默认空构造器也能使用有参数构造器两种方式创建Bean，如以下方式指定要创建的Bean类型：

使用空构造器进行定义，使用此种方式，class属性指定的类必须有空构造器

```
1. <bean name="bean1" class="cn.javass.spring.chapter2.HelloImpl2"/>
```

使用有参数构造器进行定义，使用此中方式，可以使用< constructor-arg >标签指定构造器参数值，其中index表示位置，value表示常量值，也可以指定引用，指定引用使用ref来引用另一个Bean定义，后边会详细介绍：

```
1. <bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">
2. <!-- 指定构造器参数 -->
3. <constructor-arg index="0" value="Hello Spring!"/>
4. </bean>
```

知道如何配置了，让我们做个例子的例子来实践一下吧：

（1）准备Bean class(HelloImpl2.java)，该类有一个空构造器和一个有参构造器：

```
1. package cn.javass.spring.chapter2;
2. public class HelloImpl2 implements HelloApi {
3. private String message;
4. public HelloImpl2() {
5. this.message = "Hello World!";
6. }
7. Public HelloImpl2(String message) {
8. this.message = message;
9. }
10. @Override
11. public void sayHello() {
12. System.out.println(message);
13. }
14. }
```

（2）在配置文件(resources/chapter2/instantiatingBean.xml)配置Bean定义，如下所示：

```
1. <!--使用默认构造参数-->
2. <bean name="bean1" class="cn.javass.spring.chapter2.HelloImpl2"/>
3. <!--使用有参数构造参数-->
```

```
4. <bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">
5. <!-- 指定构造器参数 -->
6. <constructor-arg index="0" value="Hello Spring!"/>
7. </bean>
```

（3）配置完了，让我们写段测试代码（InstantiatingContainerTest）来看下是否工作吧：

```
1. @Test
2. public void testInstantiatingBeanByConstructor() {
3. //使用构造器
4. BeanFactory beanFactory =
5. new ClassPathXmlApplicationContext("chapter2/instantiatingBean.xml");
6. HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
7. bean1.sayHello();
8. HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
9. bean2.sayHello();
10. }
```

二、使用静态工厂方式实例化Bean，使用这种方式除了指定必须的class属性，还要指定factory-method属性来指定实例化Bean的方法，而且使用静态工厂方法也允许指定方法参数，spring IoC容器将调用此属性指定的方法来获取Bean，配置如下所示：

（1）先来看看静态工厂类代码吧HelloApiStaticFactory：

```
1. public class HelloApiStaticFactory {
2. //工厂方法
3. public static HelloApi newInstance(String message) {
4. //返回需要的Bean实例
5. return new HelloImpl2(message);
6. }
7. }
```

（2）静态工厂写完了，让我们在配置文件(resources/chapter2/instantiatingBean.xml)配置Bean定义：

```
1. <!-- 使用静态工厂方法 -->
2. <bean id="bean3" class="cn.javass.spring.chapter2.HelloApiStaticFactory" factory-
   method="newInstance">
3. <constructor-arg index="0" value="Hello Spring!"/>
4. </bean>
```

（3）配置完了，写段测试代码来测试一下吧，InstantiatingBeanTest：

```
1. @Test
```

```
2. public void testInstantiatingBeanByStaticFactory() {
3. //使用静态工厂方法
4. BeanFactory beanFactory =
5. new ClassPathXmlApplicationContext("chaper2/instantiatingBean.xml");
6. HelloApi bean3 = beanFactory.getBean("bean3", HelloApi.class);
7. bean3.sayHello();
8. }
```

三、使用实例工厂方法实例化Bean，使用这种方式不能指定class属性，此时必须使用factory-bean属性来指定工厂Bean，factory-method属性指定实例化Bean的方法，而且使用实例工厂方法允许指定方法参数，方式和使用构造器方式一样，配置如下：

（1）实例工厂类代码（HelloApiInstanceFactory.java）如下：

```
1. package cn.javass.spring.chapter2;
2. public class HelloApiInstanceFactory {
3. public HelloApi newInstance(String message) {
4. return new HelloImpl2(message);
5. }
6. }
```

（2）让我们在配置文件(resources/chapter2/instantiatingBean.xml)配置Bean定义：

```
1. <!—1、定义实例工厂Bean -->
2. <bean id="beanInstanceFactory"
3. class="cn.javass.spring.chapter2.HelloApiInstanceFactory"/>
4. <!—2、使用实例工厂Bean创建Bean -->
5. <bean id="bean4"
6. factory-bean="beanInstanceFactory"
7. factory-method="newInstance">
8. <constructor-arg index="0" value="Hello Spring!"></constructor-arg>
9. </bean>
```

（3）测试代码InstantiatingBeanTest：

```
1. @Test
2. public void testInstantiatingBeanByInstanceFactory() {
3. //使用实例工厂方法
4. BeanFactory beanFactory =
5. new ClassPathXmlApplicationContext("chapter2/instantiatingBean.xml");
6. HelloApi bean4 = beanFactory.getBean("bean4", HelloApi.class);
7. bean4.sayHello();
8. }
```

通过以上例子我们已经基本掌握了如何实例化Bean了，大家是否注意到？这三种方式只是配置不一样，从获取方式看完全一样，没有任何不同。这也是Spring IoC的魅力，Spring IoC帮你创建Bean，我们只管使用就可以了，是不是很简单。

2.3.5 小结

到此我们已经讲完了Spring IoC基础部分，包括IoC容器概念，如何实例化容器，Bean配置、命名及实例化，Bean获取等等。不知大家是否注意到到目前为止，我们只能通过简单的实例化Bean，没有涉及Bean之间关系。接下来一章让我们进入配置Bean之间关系章节，也就是依赖注入。

【第三章】 DI 之 3.1 DI的配置使用 ——跟我学spring3

3.1.1 依赖和依赖注入

传统应用程序设计中所说的依赖一般指“类之间的关系”，那先让我们复习一下类之间的关系：

泛化：表示类与类之间的继承关系、接口与接口之间的继承关系；

实现：表示类对接口的实现；

依赖：当类与类之间有使用关系时就属于依赖关系，不同于关联关系，依赖不具有“拥有关系”，而是一种“相识关系”，只在某个特定地方（比如某个方法体内）才有关系。

关联：表示类与类或类与接口之间的依赖关系，表现为“拥有关系”；具体到代码可以用实例变量来表示；

聚合：属于是关联的特殊情况，体现部分-整体关系，是一种弱拥有关系；整体和部分可以有不一样的生命周期；是一种弱关联；

组合：属于是关联的特殊情况，也体现了体现部分-整体关系，是一种强“拥有关系”；整体与部分有相同的生命周期，是一种强关联；

Spring IoC容器的依赖有两层含义：**Bean**依赖容器和容器注入**Bean**的依赖资源：

Bean依赖容器：也就是说Bean要依赖于容器，这里的依赖是指容器负责创建Bean并管理Bean的生命周期，正是由于由容器来控制创建Bean并注入依赖，也就是控制权被反转了，这也正是IoC名字的由来，此处的有依赖是指**Bean**和容器之间的依赖关系。

容器注入**Bean**的依赖资源：容器负责注入Bean的依赖资源，依赖资源可以是Bean、外部文件、常量数据等，在Java中都反映为对象，并且由容器负责组装Bean之间的依赖关系，此处的依赖是指**Bean**之间的依赖关系，可以认为是传统类与类之间的**“**关联**”**、**“**聚合**”**、**“**组合**”**关系。

为什么要应用依赖注入，应用依赖注入能给我们带来哪些好处呢？

动态替换**Bean**依赖对象，程序更灵活：替换Bean依赖对象，无需修改源文件：应用依赖注入后，由于可以采用配置文件方式实现，从而能随时动态的替换Bean的依赖对象，无需修改java源文件；

更好实践面向接口编程，代码更清晰：在Bean中只需指定依赖对象的接口，接口定义依赖对象完成的功能，通过容器注入依赖实现；

更好实践优先使用对象组合，而不是类继承：因为IoC容器采用注入依赖，也就是组合对象，从而更好的实践对象组合。

- 采用对象组合，Bean的功能可能由几个依赖Bean的功能组合而成，其Bean本身可能只提供少许功能或根本无任何功能，全部委托给依赖Bean，对象组合具有动态性，能更方便的替换掉依赖Bean，从而改变Bean功能；
- 而如果采用类继承，Bean没有依赖Bean，而是采用继承方式添加新功能，，而且功能是在编译时就确定了，不具有动态性，而且采用类继承导致Bean与子Bean之间高度耦合，难以复用。

增加**Bean**可复用性：依赖于对象组合，Bean更可复用且复用更简单；

降低**Bean**之间耦合：由于我们完全采用面向接口编程，在代码中没有直接引用Bean依赖实现，全部引用接口，而且不会出现显示的创建依赖对象代码，而且这些依赖是由容器来注入，很容易替换依赖实现类，从而降低Bean与依赖之间耦合；

代码结构更清晰：要应用依赖注入，代码结构要按照规约方式进行书写，从而更好的应用一些最佳实践，因此代码结构更清晰。

从以上我们可以看出，其实依赖注入只是一种装配对象的手段，设计的类结构才是基础，如果设计的类结构不支持依赖注入，Spring IoC容器也注入不了任何东西，从而从根本上说“如何设计好类结构才是关键，依赖注入只是一种装配对象手段”。

前边IoC一章我们已经了解了Bean依赖容器，那容器如何注入Bean的依赖资源，Spring IoC容器注入依赖资源主要有以下两种基本实现方式：

构造器注入：就是容器实例化Bean时注入那些依赖，通过在在Bean定义中指定构造器参数进行注入依赖，包括实例工厂方法参数注入依赖，但静态工厂方法参数不允许注入依赖；

setter注入：通过setter方法进行注入依赖；

方法注入：能通过配置方式替换掉Bean方法，也就是通过配置改变Bean方法 功能。

我们已经知道注入实现方式了，接下来让我们来看看具体配置吧。

3.1.2 构造器注入

使用构造器注入通过配置构造器参数实现，构造器参数就是依赖。除了构造器方式，还有静态工厂、实例工厂方法可以进行构造器注入。如图3-1所示：

通过容器构造器依赖注入实例化

<bean class="...HelloImpl3">
<constructor-arg index="0" value="Hello!"/>
<constructor-arg index="1" value="1"/>
</bean>

1、实例化
2、设置参数
3、设置参数

传统实例化方式

HelloApi api = new HelloImpl3(
"Hello!",
);

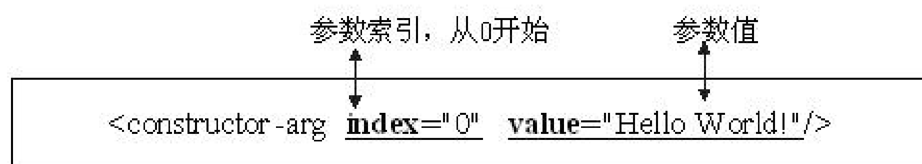
图3-1 实例化

构造器注入可以根据参数索引注入、参数类型注入或Spring3支持的参数名注入，但参数名注入是有限制的，需要使用在编译程序时打开调试模式（即在编译时使用“javac –g:vars”在class文件中生成变量调试信息，默认是不包含变量调试信息的，从而能获取参数名字，否则获取不到参数名字）或在构造器上使用@ConstructorProperties（java.beans.ConstructorProperties）注解来指定参数名。

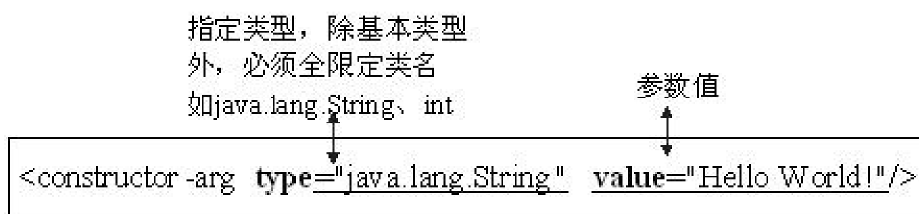
首先让我们准备测试构造器类HelloImpl3.java，该类只有一个包含两个参数的构造器：

```
1.  package cn.javass.spring.chapter3.helloworld;
2.  public class HelloImpl3 implements HelloApi {
3.  private String message;
4.  private int index;
5.  //@java.beans.ConstructorProperties({"message", "index"})
6.  public HelloImpl3(String message, int index) {
7.  this.message = message;
8.  this.index = index;
9.  }
10. @Override
11. public void sayHello() {
12. System.out.println(index + ":" + message);
13. }
14. }
```

一、根据参数索引注入，使用标签“<constructor-arg index="1" value="1"/>”来指定注入的依赖，其中“index”表示索引，从0开始，即第一个参数索引为0，“value”来指定注入的常量值，配置方式如下：



二、根据参数类型进行注入，使用标签“<constructor-arg type="java.lang.String" value="Hello World!"/>”来指定注入的依赖，其中“type”表示需要匹配的参数类型，可以是基本类型也可以是其他类型，如“int”、“java.lang.String”，“value”来指定注入的常量值，配置方式如下：



三、根据参数名进行注入，使用标签“<constructor-arg name="message" value="Hello World!"/>”来指定注入的依赖，其中“name”表示需要匹配的参数名字，“value”来指定注入的常量值，配置方式如下：



四、让我们来用具体的例子来看一下构造器注入怎么使用吧。

（1）首先准备Bean类，在此我们就使用“HelloImpl3”这个类。

（2）有了Bean类，接下来要进行Bean定义配置，我们需要配置三个Bean来完成上述三种依赖注入测试，其中Bean ”byIndex”是通过索引注入依赖；Bean ”byType”是根据类型进行注入依赖；Bean ”byName”是根据参数名字进行注入依赖，具体配置文件（resources/chapter3/constructorDependencyInject.xml）如下：

```
1. <!-- 通过构造器参数索引方式依赖注入 -->
2. <bean id="byIndex" class="cn.javass.spring.chapter3.HelloImpl3">
3. <constructor-arg index="0" value="Hello World!"/>
4. <constructor-arg index="1" value="1"/>
5. </bean>
6. <!-- 通过构造器参数类型方式依赖注入 -->
7. <bean id="byType" class="cn.javass.spring.chapter3.HelloImpl3">
8. <constructor-arg type="java.lang.String" value="Hello World!"/>
9. <constructor-arg type="int" value="2"/>
10. </bean>
11. <!-- 通过构造器参数名称方式依赖注入 -->
12. <bean id="byName" class="cn.javass.spring.chapter3.HelloImpl3">
13. <constructor-arg name="message" value="Hello World!"/>
14. <constructor-arg name="index" value="3"/>
15. </bean>
```

（3）配置完毕后，在测试之前，因为我们使用了通过构造器参数名字注入方式，请确保编译时class文件包含“变量信息”，具体查看编译时是否包含“变量调试信息”请右击项目，在弹出的对话框选择属性；然后在弹出的对话框选择“Java Compiler”条目，在“Class 文件 生成”框中选择“添加变量信息到Class文件（调试器使用）”，具体如图3-2：

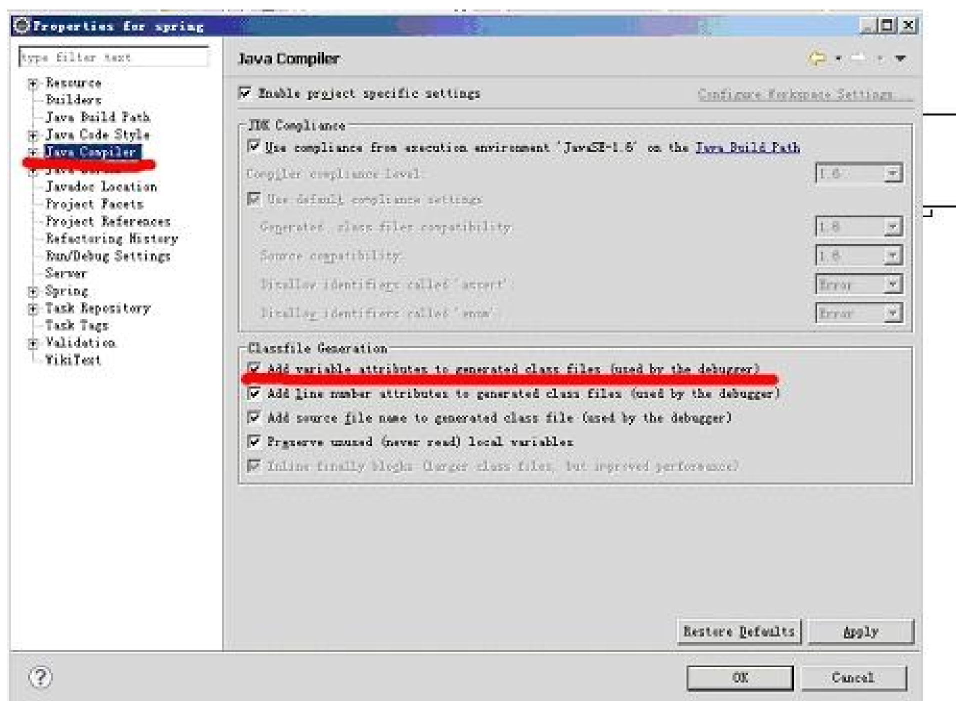


图3-2 编译时打开“添加变量信息选项”

（4）接下来让我们测试一下配置是否工作，具体测试代码（cn.javass.spring.chapter3.DependencyInjectTest）如下：

```
1.  @Test
2.  public void testConstructorDependencyInjectTest() {
3.  BeanFactory beanFactory = new
    ClassPathXmlApplicationContext("chapter3/constructorDependencyInject.xml");
4.  //获取根据参数索引依赖注入的Bean
5.  HelloApi byIndex = beanFactory.getBean("byIndex", HelloApi.class);
6.  byIndex.sayHello();
7.  //获取根据参数类型依赖注入的Bean
8.  HelloApi byType = beanFactory.getBean("byType", HelloApi.class);
9.  byType.sayHello();
10. //获取根据参数名字依赖注入的Bean
11. HelloApi byName = beanFactory.getBean("byName", HelloApi.class);
12. byName.sayHello();
13. }
```

通过以上测试我们已经会基本的构造器注入配置了，在测试通过参数名字注入时，除了可以使用以上方式，还可以通过在构造器上添加@java.beans.ConstructorProperties({"message", "index"})注解来指定参数名字，在HelloImpl3构造器上把注释掉的“ConstructorProperties”打开就可以了，这个就留给大家做练习，自己配置然后测试一下。

五、大家已经会了构造器注入，那让我们再看一下静态工厂方法注入和实例工厂注入吧，其实它们注入配置是完全一样，在此我们只示范一下静态工厂注入方式和实例工厂方式配置，测试就留给大家自己练习：

（1）静态工厂类

```
1. //静态工厂类
2. package cn.javass.spring.chapter3;
3. import cn.javass.spring.chapter2.helloworld.HelloApi;
4. public class DependencyInjectByStaticFactory {
5. public static HelloApi newInstance(String message, int index) {
6. return new HelloImpl3(message, index);
7. }
8. }
```

静态工厂类Bean定义配置文件（chapter3/staticFactoryDependencyInject.xml）

```
1. <bean id="byIndex"
2. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-
   method="newInstance">
3. <constructor-arg index="0" value="Hello World!"/>
4. <constructor-arg index="1" value="1"/>
5. </bean>
6. <bean id="byType"
7. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-
   method="newInstance">
8. <constructor-arg type="java.lang.String" value="Hello World!"/>
9. <constructor-arg type="int" value="2"/>
10. </bean>
11. <bean id="byName"
12. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-
   method="newInstance">
13. <constructor-arg name="message" value="Hello World!"/>
14. <constructor-arg name="index" value="3"/>
15. </bean>
```

（2）实例工厂类

```
1. //实例工厂类
2. package cn.javass.spring.chapter3;
3. import cn.javass.spring.chapter2.helloworld.HelloApi;
4. public class DependencyInjectByInstanceFactory {
5. public HelloApi newInstance(String message, int index) {
```

```
6. return new HelloImpl3(message, index);
7. }
8. }
```

实例工厂类Bean定义配置文件（chapter3/instanceFactoryDependencyInject.xml）

```
1. <bean id="instanceFactory"
2. class="cn.javass.spring.chapter3.DependencyInjectByInstanceFactory"/>
3. <bean id="byIndex"
4. factory-bean="instanceFactory" factory-method="newInstance">
5. <constructor-arg index="0" value="Hello World!"/>
6. <constructor-arg index="1" value="1"/>
7. </bean>
8. <bean id="byType"
9. factory-bean="instanceFactory" factory-method="newInstance">
10. <constructor-arg type="java.lang.String" value="Hello World!"/>
11. <constructor-arg type="int" value="2"/>
12. </bean>
13. <bean id="byName"
14. factory-bean="instanceFactory" factory-method="newInstance">
15. <constructor-arg name="message" value="Hello World!"/>
16. <constructor-arg name="index" value="3"/>
17. </bean>
```

（3）测试代码和构造器方式完全一样，只是配置文件不一样，大家只需把测试文件改一下就可以了。还有一点需要大家注意就是静态工厂方式和实例工厂方式根据参数名字注入的方式只支持通过在class文件中添加“变量调试信息”方式才能运行，ConstructorProperties注解方式不能工作，它只对构造器方式起作用，不建议使用根据参数名进行构造器注入。

3.1.3 setter注入

setter注入，是通过在通过构造器、静态工厂或实例工厂实例好Bean后，通过调用Bean类的setter方法进行注入依赖，如图3-3所示：

通过容器setter注入		传统setter方式
<bean class="...HelloImpl4">	1、实例化	HelloApi api = new HelloImpl4();
<property name="message" value="Hello"/>	2、设置属性	Api.setMessage("Hello");
<property name="index" value="1"/>	3、设置属性	Api.setIndex("Hello");
</bean>		

图3-3 setter注入方式

setter注入方式只有一种根据setter名字进行注入：



知道配置方式了，接下来先让我们来做个简单例子吧。

（1）准备测试类HelloImpl4，需要两个setter方法“setMessage”和“setIndex”：

```
1.  package cn.javass.spring.chapter3;
2.  import cn.javass.spring.chapter2.helloworld.HelloApi;
3.  public class HelloImpl4 implements HelloApi {
4.  private String message;
5.  private int index;
6.  //setter方法
7.  public void setMessage(String message) {
8.  this.message = message;
9.  }
10. public void setIndex(int index) {
11. this.index = index;
12. }
13. @Override
14. public void sayHello() {
15. System.out.println(index + ":" + message);
16. }
17. }
```

（2）配置Bean定义，具体配置文件（resources/chapter3/setterDependencyInject.xml）片段如下：

```
1.  <!-- 通过setter方式进行依赖注入 -->
2.  <bean id="bean" class="cn.javass.spring.chapter3.HelloImpl4">
3.  <property name="message" value="Hello World!"/>
4.  <property name="index">
5.  <value>1</value>
6.  </property>
7.  </bean>
```

（3）该写测试进行测试一下是否满足能工作了，其实测试代码一点没变，变的是配置：

```
1. @Test
2. public void testSetterDependencyInject() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter3/setterDependencyInject.xml");
5. HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
6. bean.sayHello();
7. }
```

知道如何配置了，但Spring如何知道setter方法？如何将值注入进去的呢？其实方法名是要遵守约定的，setter注入的方法名要遵循“JavaBean getter/setter 方法命名约定”：

JavaBean：是本质就是一个POJO类，但具有一下限制：

该类必须要有公共的无参构造器，如public HelloImpl4() {}；

属性为**private**访问级别，不建议public，如private String message;

属性必要时通过一组**setter**（修改器）和**getter**（访问器）方法来访问；

setter方法，以**“set”** 开头，后跟首字母大写的属性名，如“setMesssage”,简单属性一般只有一个方法参数，方法返回值通常为“void”;

getter方法，一般属性以**“get”**开头，对于**boolean**类型一般以**“is”**开头，后跟首字母大写的属性名，如“getMesssage”，“isOk”；

还有一些其他特殊情况，比如属性有连续两个大写字母开头，如**“URL”,**则**setter/getter**方法为：**“setURL”**和**“getURL”**，其他一些特殊情况请参看“Java Bean”命名规范。

3.1.4 注入常量

注入常量是依赖注入中最简单的。配置方式如下所示：

```
1. <property name="message" value="Hello World!"/>
2. 或
3. <property name="index"><value>1</value></property><span class="Apple-style-span" style="font-size: 14px; white-space: normal; background-color: #ffffff;"> </span>
```

以上两种方式都可以，从配置来看第一种更简洁。注意此处“value”中指定的全是字符串，由Spring容器将此字符串转换成属性所需要的类型，如果转换出错，将抛出相应的异常。

Spring容器目前能对各种基本类型把配置的String参数转换为需要的类型。

注：Spring类型转换系统对于boolean类型进行了容错处理，除了可以使用“true/false”标准的Java值进行注入，还能使用“yes/no”、“on/off”、“1/0”来代表“眞/假”，所以大家在学习或工作中遇到这种类似问题不要觉得是人家配置错了，而是Spring容错做的非常好。

1. 测试类

```
2. public class BooleanTestBean {
3. private boolean success;
4. public void setSuccess(boolean success) {
5. this.success = success;
6. }
7. public boolean isSuccess() {
8. return success;
9. }
10. }
11. 配置文件（chapter3/booleanInject.xml）片段：
12. <!-- boolean参数值可以用on/off -->
13. <bean id="bean2" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
14. <property name="success" value="on"/>
15. </bean>
16. <!-- boolean参数值可以用yes/no -->
17. <bean id="bean3" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
18. <property name="success" value="yes"/>
19. </bean>
20. <!-- boolean参数值可以用1/0 -->
21. <bean id="bean4" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
22. <property name="success" value="1"/>
23. </bean>
```

3.1.5 注入Bean ID

用于注入Bean的ID，ID是一个常量不是引用，且类似于注入常量，但提供错误验证功能，配置方式如下所示：

```
1. <property name="id"><idref bean="bean1"/></property>
```

```
1. <property name="id"><idref local="bean2"/></property>
```

两种方式都可以，上述配置本质上在运行时等于如下方式

```
1. <bean id="bean1" class="……"/>
2. <bean id="idrefBean1" class="……">
3. <property name="id" value ="bean1"/>
4. </bean>
```

第二种方式（<idref bean="bean1"/>）可以在容器初始化时校验被引用的Bean是否存在，如果不存在将抛出异常，而第一种方式（<idref local="bean2"/>）只有在Bean实际使用时才能发现传入的Bean的ID是否正确，可能发生不可预料的错误。因此如果想注入Bean的ID，推荐使用第二种方式。

接下来学习一下如何使用吧：

首先定义测试Bean：

```
1. package cn.javass.spring.chapter3.bean
2. public class IdRefTestBean {
3. private String id;
4. public String getId() {
5. return id;
6. }
7. public void setId(String id) {
8. this.id = id;
9. }
10. }
```

其次定义配置文件（**chapter3/idRefInject.xml**）：

```
1. <bean id="bean1" class="java.lang.String">
2. <constructor-arg index="0" value="test"/>
3. </bean>
4. <bean id="bean2" class="java.lang.String">
5. <constructor-arg index="0" value="test"/>
6. </bean>
```

```
1. <bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
2. <property name="id"><idref bean="bean1"/></property>
3. </bean>
4. <bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
5. <property name="id"><idref local="bean2"/></property>
6. </bean>
```

从配置中可以看出，注入的Bean的ID是一个java.lang.String类型，即字符串类型，因此注入的同样是常量，只是具有校验功能。

<idref bean="……"/>将在容器初始化时校验注入的ID对于的Bean是否存在，如果不存在将抛出异常。

<idref local="……"/>将在XML解析时校验注入的ID对于的Bean在当前配置文件中是否存在，如果不存在将抛出异常，它不同于<idref bean="……"/>，<idref local="……"/>是校验发生在XML解析式而非容器初始化时，且只检查当前配置文件中是否存在相应的Bean。

3.1.6 注入集合、数组和字典

Spring不仅能注入简单类型数据，还能注入集合（Collection、无序集合Set、有序集合List）类型、数组(Array)类型、字典(Map)类型数据、Properties类型数据，接下来就让我们一个个看看如何注入这些数据类型的数据。

一、注入集合类型：包括Collection类型、Set类型、List类型数据：

（**1**）**List**类型：需要使用<list>标签来配置注入，其具体配置如下：

1、可选的"value-type"属性，表示列表中条目的数据的类型，比如value-type="java.lang.String"表示列表需要条目为String数据类型；

2、也可以采用泛型，Spring能根据泛型数据自动检测到List里条目的数据类型，比如java.util.List<String>，Spring能自动识别列表需要条目为String数据类型；

3、如果既没有指定"value-type"属性List也不是泛型的则默认就是String类型；

```
<bean ……>
   <property name="values">
      <list value-type="java.lang.String"  merge="default">
         <value>1</value>
         <value>2</value>
         <value>3</value>
      </list>
   </property>
</bean>
```

指定列表条目值

可选的merge属性，后边再介绍，用于父子Bean情况是否合并list条目。

让我们来写个测试来练习一下吧：

准备测试类：

```
1.  package cn.javass.spring.chapter3.bean;
2.  import java.util.List;
3.  public class ListTestBean {
4.  private List<String> values;
5.  public List<String> getValues() {
6.  return values;
7.  }
8.  public void setValues(List<String> values) {
9.  this.values = values;
10. }
11. }
```

进行**Bean**定义，在配置文件（**resources/chapter3/listInject.xml**）中配置**list**注入：

```
1.  <bean id="listBean" class="cn.javass.spring.chapter3.bean.ListTestBean">
2.  <property name="values">
3.  <list>
```

```
4. <value>1</value>
5. <value>2</value>
6. <value>3</value>
7. </list>
8. </property>
9. </bean>
```

测试代码：

```
1. @Test
2. public void testListInject() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter3/listInject.xml");
5. ListTestBean listBean = beanFactory.getBean("listBean", ListTestBean.class);
6. System.out.println(listBean.getValues().size());
7. Assert.assertEquals(3, listBean.getValues().size());
8. }
```

（**2**）**Set**类型：需要使用<set>标签来配置注入，其配置参数及含义和<lsit>标签完全一样，在此就不阐述了：

准备测试类：

```
1. package cn.javass.spring.chapter3.bean;
2. import java.util.Collection;
3. public class CollectionTestBean {
4. private Collection<String> values;
5. public void setValues(Collection<String> values) {
6. this.values = values;
7. }
8. public Collection<String> getValues() {
9. return values;
10. }
11. }
```

进行**Bean**定义，在配置文件（**resources/chapter3/listInject.xml**）中配置**list**注入：

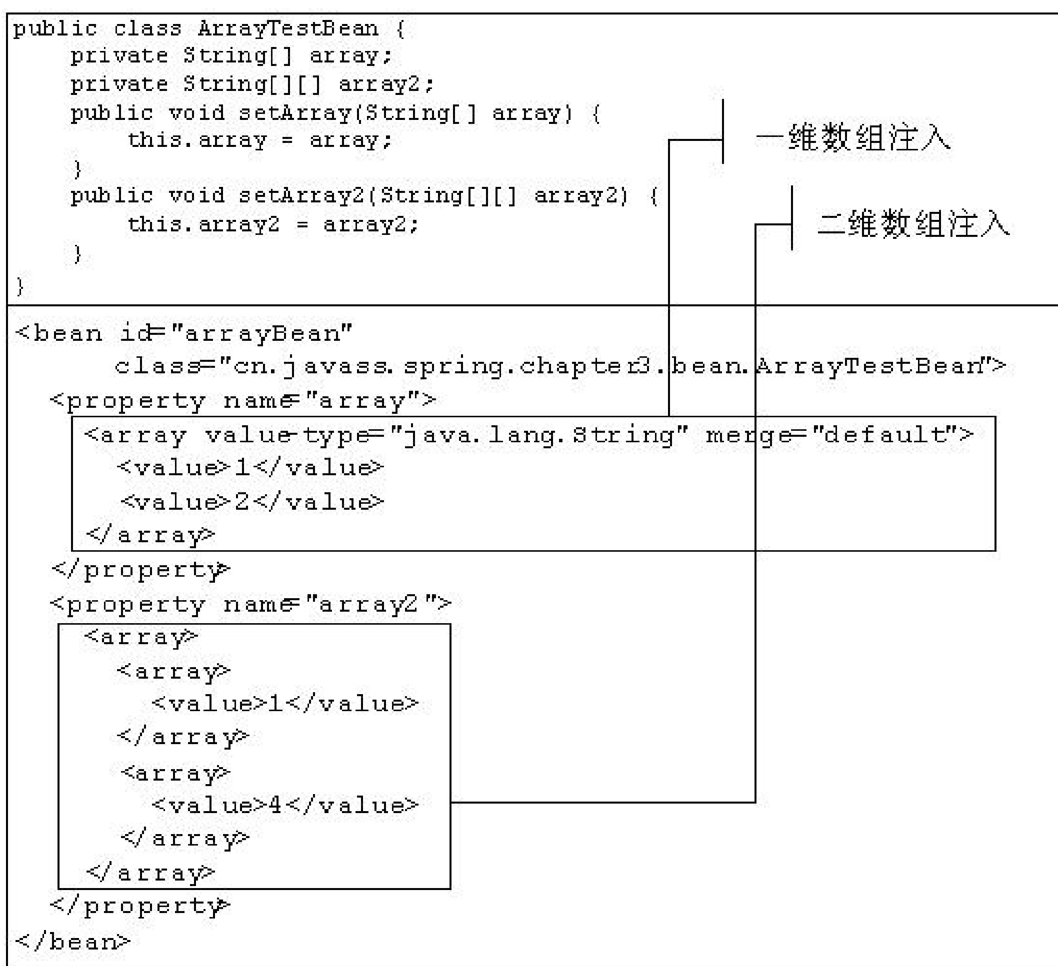
```
1. <bean id="setBean" class="cn.javass.spring.chapter3.bean.SetTestBean">
2. <property name="values">
3. <set>
4. <value>1</value>
5. <value>2</value>
6. <value>3</value>
7. </set>
```

8. </property>
9. </bean>

具体测试代码就不写了，和listBean测试代码完全一样。

（**2**）**Collection**类型：因为Collection类型是Set和List类型的基类型，所以使用<set>或<list>标签都可以进行注入，配置方式完全和以上配置方式一样，只是将测试类属性改成“Collection”类型，如果配置有问题，可参考cn.javass.spring.chapter3.DependencyInjectTest测试类中的testCollectionInject测试方法中的代码。

二、注入数组类型：需要使用<array>标签来配置注入，其中标签属性“value-type”和“merge”和<list>标签含义完全一样，具体配置如下：



```
public class ArrayTestBean {
    private String[] array;
    private String[][] array2;
    public void setArray(String[] array) {
        this.array = array;
    }
    public void setArray2(String[][] array2) {
        this.array2 = array2;
    }
}
```

```
<bean id="arrayBean"
      class="cn.javass.spring.chapter3.bean.ArrayTestBean">
  <property name="array">
    <array value-type="java.lang.String" merge="default">
      <value>1</value>
      <value>2</value>
    </array>
  </property>
  <property name="array2">
    <array>
      <array>
        <value>1</value>
      </array>
      <array>
        <value>4</value>
      </array>
    </array>
  </property>
</bean>
```


如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的testArrayInject测试方法中的代码。

三、注入字典（**Map**）类型：字典类型是包含键值对数据的数据结构，需要使用<map>标签来配置注入，其属性“key-type”和“value-type”分别指定“键”和“值”的数据类型，其含义和<list>标签的“value-type”含义一样，在此就不罗嗦了，并使用<key>子标签来指定键数据，<value>子标签来指定键对应的值数据，具体配置如下：

```
//对应的类文件代码
public class MapTestBean{
    private Map<String, String> values;
    //对应的setter/getter方法
    public void setValues(Map<String, String> values) {
        this.values = values;
    }
}

    <bean id="mapBean"
class="cn.javass.spring.chapter3.bean.MapTestBean">
        <property name="values">
            <map  key-type="java.lang.String"
                  value-type="java.lang.String">
                <entry>
                    <key><value>1</value></key>
                    <value>11</value>
                </entry>

                <entry key="2" value="22"/>
            </map>
        </property>
    </bean>
```


如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的testMapInject测试方法中的代码。

四、**Properties**注入：Spring能注入java.util.Properties类型数据，需要使用<props>标签来配置注入，键和值类型必须是String，不能变，子标签<prop key=”键”>值</prop>来指定键值对，具体配置如下：

```
public class PropertiesTestBean{
    private Properties values;
    public void setValues(Properties values) {
        this.values = values;
    }
}
 <bean id="propertiesBean"
  class="cn.javass.spring.chapter3.bean.PropertiesTestBean">
  <property name="values">
    <props value-type="int" merge="default">
      <prop key="1">1sss</prop>
      <prop key="2">2</prop>
    </props>
  </property>
</bean>
```

```
public class PropertiesTestBean{
    private Properties values;
    public void setValues(Properties values) {
        this.values = values;
    }
}
                                                    配置
<bean id="propertiesBean"
  class="cn.javass.spring.chapter3.bean.PropertiesTestBean">
  <property name="values">
    <value>
      1=11
      2=22;         分隔符可以是“换行”、“;”、“,”
      3=33,         不建议使用该配置方式，应该优先选择第一种配置方式
      4=44
    </value>
  </property>
</bean>
```

如果练习时遇到配置问题，可以参考cn.javass.spring.chapter3.DependencyInjectTest测试类中的testPropertiesInject测试方法中的代码。

到此我们已经把简单类型及集合类型介绍完了，大家可能会问怎么没见注入“Bean之间关系”的例子呢？接下来就让我们来讲解配置Bean之间依赖关系，也就是注入依赖Bean。

3.1.7 引用其它Bean

上边章节已经介绍了注入常量、集合等基本数据类型和集合数据类型，本小节将介绍注入依赖Bean及注入内部Bean。

引用其他Bean的步骤与注入常量的步骤一样，可以通过构造器注入及setter注入引用其他Bean，只是引用其他Bean的注入配置稍微变化了一下：可以将“<constructor-arg index="0" value="Hello World!"/>”和“<property name="message" value="Hello World!"/>”中的value属性替换成bean属性，其中bean属性指定配置文件中的其他Bean的id或别名。另一种是把<value>标签替换为<.ref bean="beanName">，bean属性也是指定配置文件中的其他Bean的id或别名。那让我们看一下具体配置吧：

一、构造器注入方式：

（1）通过” <constructor-arg>”标签的ref属性来引用其他Bean，这是最简化的配置：

```
<constructor-arg index="0" value="Hello!"/>    注入常量
<constructor-arg index="0" ref="bean"   />     注入Bean
<bean id="bean" class="……"/>   引用”bean”
```

（2）通过” <constructor-arg>”标签的子<ref>标签来引用其他Bean，使用bean属性来指定引用的Bean：

```
注入常量
<constructor-arg index="0"><value>Hello!</value></constructor-arg>
注入Bean
<constructor-arg index="0"><ref bean="bean"/></constructor-arg>
<beanid="bean" class="……"/>    引用" bean"
```

二、**setter**注入方式：

（1）通过” <property>”标签的ref属性来引用其他Bean，这是最简化的配置：

```
<property name="message" value="Hello World!"/>    注入常量
<property name="message" ref="bean"/>    注入Bean
<beanid="bean" class="……"/>    引用" bean"
```

（2）通过” <property>”标签的子<ref>标签来引用其他Bean，使用bean属性来指定引用的Bean：

```
<property name="message"><value>HelloWorld!</value></property>    注入常量
<property name="message"><ref bean="beanName"/></property>    注入Bean
<bean id="bean" class="……"/>    引用" bean"
```

三、接下来让我们用个具体例子来讲解一下具体使用吧：

（1）首先让我们定义测试引用Bean的类，在此我们可以使用原有的HelloApi实现，然后再定义一个装饰器来引用其他Bean，具体装饰类如下：

1. package cn.javass.spring.chapter3.bean;
2. import cn.javass.spring.chapter2.helloworld.HelloApi;
3. public class HelloApiDecorator implements HelloApi {
4. private HelloApi helloApi;
5. //空参构造器
6. public HelloApiDecorator() {
7. }
8. //有参构造器

```
9.  public HelloApiDecorator(HelloApi helloApi) {
10. this.helloApi = helloApi;
11. }
12. public void setHelloApi(HelloApi helloApi) {
13. this.helloApi = helloApi;
14. }
15. @Override
16. public void sayHello() {
17. System.out.println("==========装饰一下===========");
18. helloApi.sayHello();
19. System.out.println("==========装饰一下===========");
20. }
21. }
```

（2）定义好了测试引用Bean接下来该在配置文件(resources/chapter3/beanInject.xml)进行配置Bean定义了，在此将演示通过构造器及setter方法方式注入依赖Bean：

```
1.  <!-- 定义依赖Bean -->
2.  <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
3.  <!-- 通过构造器注入 -->
4.  <bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
5.  <constructor-arg index="0" ref="helloApi"/>
6.  </bean>
7.  <!-- 通过构造器注入 -->
8.  <bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
9.  <property name="helloApi"><ref bean=" helloApi"/></property>
10. </bean>
```

（3）测试一下吧，测试代码(cn.javass.spring.chapter3.DependencyInjectTest)片段如下：

```
1.  @Test
2.  public void testBeanInject() {
3.  BeanFactory beanFactory =
4.  new ClassPathXmlApplicationContext("chapter3/beanInject.xml");
5.  //通过构造器方式注入
6.  HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
7.  bean1.sayHello();
8.  //通过setter方式注入
9.  HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
10. bean2.sayHello();
11. }
```

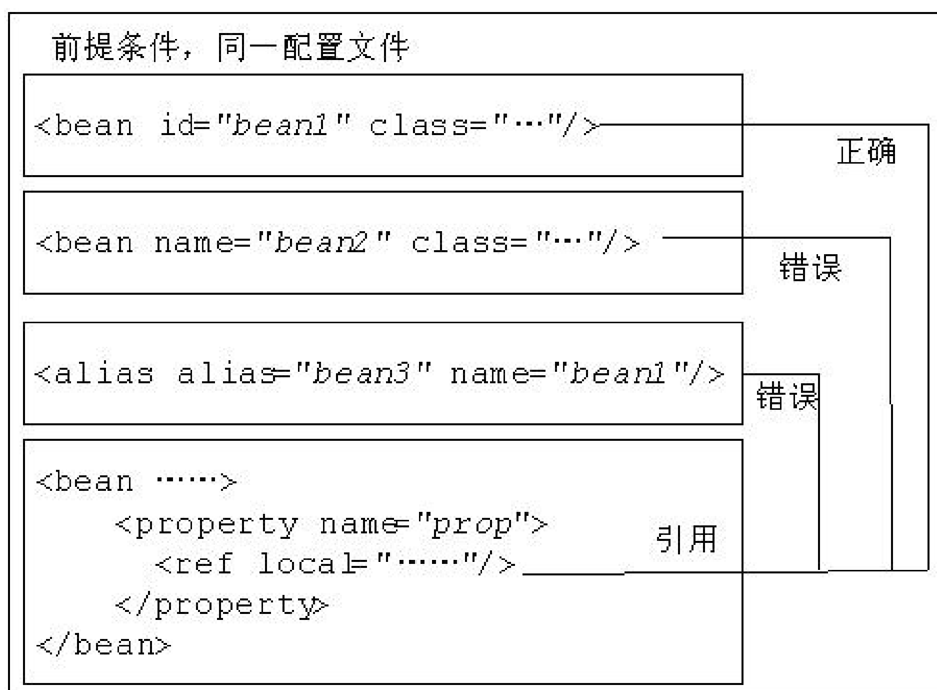
四、其他引用方式：除了最基本配置方式以外，Spring还提供了另外两种更高级的配置方式，<ref local=""/>和<ref parent=""/>：

（1）<ref local=""/>配置方式：用于引用通过<bean id="beanName">方式中通过id属性指定的Bean，它能利用XML解析器的验证功能在读取配置文件时来验证引用的Bean是否存在。因此如果在当前配置文件中有相互引用的Bean可以采用<ref local>方式从而如果配置错误能在开发调试时就发现错误。

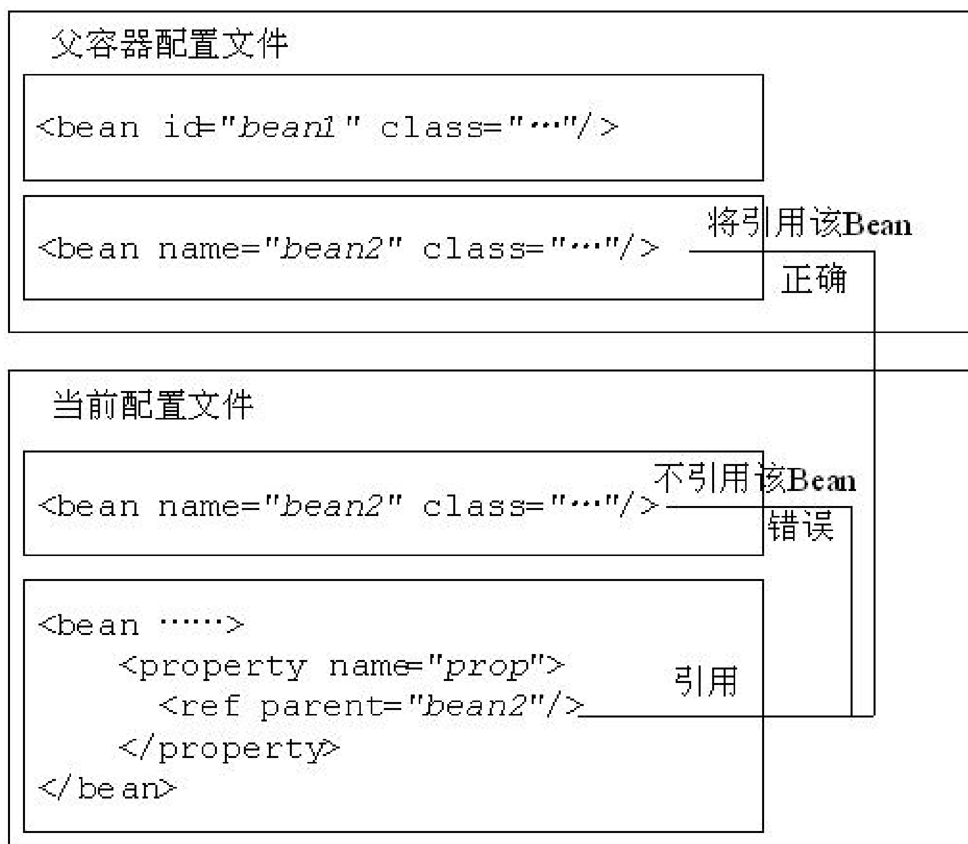
如果引用一个在当前配置文件中不存在的Bean将抛出如下异常：

org.springframework.beans.factory.xml.XmlBeanDefinitionStoreException: Line21 inXML document from class path resource [chapter3/beanInject2.xml] is invalid; nested exception is org.xml.sax.SAXParseException: cvc-id.1: There is no ID/IDREF binding for IDREF 'helloApi'.

<ref local>具体配置方式如下：



（2）<ref parent=""/>配置方式：用于引用父容器中的Bean，不会引用当前容器中的Bean，当然父容器中的Bean和当前容器的Bean是可以重名的，获取顺序是直接到父容器找。具体配置方式如下：



接下来让我们用个例子演示一下<ref local>和<ref parent>的配置过程：

首先还是准备测试类，在此我们就使用以前写好的HelloApiDecorator和HelloImpl4类；其次进行Bean定义，其中当前容器bean1引用本地的"helloApi"，而"bean2"将引用父容器的"helloApi"，配置如下：

```
1. <!-- sources/chapter3/parentBeanInject.xml表示父容器配置-->
2. <!--注意此处可能子容器也定义一个该Bean-->
3. <bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
4. <property name="index" value="1"/>
5. <property name="message" value="Hello Parent!"/>
6. </bean>
```

```
1. <!-- sources/chapter3/localBeanInject.xml表示当前容器配置-->
2. <!-- 注意父容器中也定义了id 为 helloApi的Bean -->
3. <bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
4. <property name="index" value="1"/>
5. <property name="message" value="Hello Local!"/>
6. </bean>
7. <!-- 通过local注入 -->
8. <bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
9. <constructor-arg index="0"><ref local="helloApi"/></constructor-arg>
10. </bean>
```

```
11. <!-- 通过parent注入 -->
12. <bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
13. <property name="helloApi"><ref parent="helloApi"/></property>
14. </bean>
```

（3）写测试类测试一下吧，具体代码片段如下：

```
1. @Test
2. public void testLocalAndparentBeanInject() {
3. //初始化父容器
4. ApplicationContext parentBeanContext =
5. new ClassPathXmlApplicationContext("chapter3/parentBeanInject.xml");
6. //初始化当前容器
7. ApplicationContext beanContext = new ClassPathXmlApplicationContext(
8. new String[] {"chapter3/localBeanInject.xml"}, parentBeanContext);
9. HelloApi bean1 = beanContext.getBean("bean1", HelloApi.class);
10. bean1.sayHello();//该Bean引用local bean
11. HelloApi bean2 = beanContext.getBean("bean2", HelloApi.class);
12. bean2.sayHello();//该Bean引用parent bean
13. }
```

“bean1”将输出“Hello Local!”表示引用当前容器的Bean，”bean2”将输出“Hello Paren!”，表示引用父容器的Bean，如配置有问题请参考cn.javass.spring.chapter3.DependencyInjectTest中的testLocalAndparentBeanInject测试方法。

3.1.8 内部Bean定义

内部Bean就是在<property>或<constructor-arg>内通过<bean>标签定义的Bean，该Bean不管是否指定id或name，该Bean都会有唯一的匿名标识符，而且不能指定别名，该内部Bean对其他外部Bean不可见，具体配置如下：



（1）让我们写个例子测试一下吧，具体配置文件如下：

```
1. <bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
2. <property name="helloApi">
3. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
4. </property>
5. </bean>
```

（2）测试代码（cn.javass.spring.chapter3.DependencyInjectTest.testInnerBeanInject）：

```
1. @Test
2. public void testInnerBeanInject() {
3. ApplicationContext context =
4. new ClassPathXmlApplicationContext("chapter3/innerBeanInject.xml");
5. HelloApi bean = context.getBean("bean", HelloApi.class);
6. bean.sayHello();
7. }
```

3.1.9 处理null值

Spring通过<value>标签或value属性注入常量值，所有注入的数据都是字符串，那如何注入null值呢？通过“null”值吗？当然不是因为如果注入“null”则认为是字符串。Spring通过<null/>标签注入null值。即可以采用如下配置方式：

```
<bean class="...HelloImpl4">
    <property name="message"><null/></property>
    <property name="index" value="1"/>
</bean>
```

3.1.10 对象图导航注入支持

所谓对象图导航是指类似a.b.c这种点缀访问形式的访问或修改值。Spring支持对象图导航方式依赖注入。对象图导航依赖注入有一个限制就是比如a.b.c对象导航图注入中a和b必须为非null值才能注入c，否则将抛出空指针异常。

Spring不仅支持对象的导航，还支持数组、列表、字典、Properties数据类型的导航，对Set数据类型无法支持，因为无法导航。

数组和列表数据类型可以用array[0]、list[1]导航，注意"[]"里的必须是数字，因为是按照索引进行导航，对于数组类型注意不要数组越界错误。

字典Map数据类型可以使用map[1]、map[str]进行导航，其中"[]"里的是基本类型，无法放置引用类型。

让我们来练习一下吧。首先准备测试类，在此我们需要三个测试类，以便实现对象图导航功能演示：

NavigationC类用于打印测试代码，从而观察配置是否正确；具体类如下所示：

```
1. package cn.javass.spring.chapter3.bean;
2. public class NavigationC {
3. public void sayNavigation() {
4. System.out.println("===navigation c");
5. }
6. }
```

NavigationB类，包含对象和列表、Properties、数组字典数据类型导航，而且这些复合数据类型保存的条目都是对象，正好练习一下如何往复合数据类型中注入对象依赖。具体类如下所示：

```
1. package cn.javass.spring.chapter3.bean;
2. import java.util.List;
3. import java.util.Map;
4. import java.util.Properties;
5. public class NavigationB {
6. private NavigationC navigationC;
7. private List<NavigationC> list;
8. private Properties properties;
9. private NavigationC[] array = new NavigationC[1];
10. private Map<String, NavigationC> map;
11. //由于setter和getter方法占用太多空间，故省略，大家自己实现吧
12. }
```

NavigationA类是我们的前端类，通过对它的导航进行注入值，具体代码如下：

```
1. package cn.javass.spring.chapter3.bean;
2. public class NavigationA {
3. private NavigationB navigationB;
4. public void setNavigationB(NavigationB navigationB) {
5. this.navigationB = navigationB;
6. }
7. public NavigationB getNavigationB() {
8. return navigationB;
9. }
10. }
```

接下来该进行Bean定义配置（resources/chapter3/navigationBeanInject.xml）了，首先让我们配置一下需要被导航的数据，NavigationC和NavigationB类，其中配置NavigationB时注意要确保比如array字段不为空值，这就需要或者在代码中赋值如“NavigationC[] array = new NavigationC[1];”，或者通过配置文件注入如“<list></list>”注入一个不包含条目的列表。具体配置如下：

```
1. <bean id="c" class="cn.javass.spring.chapter3.bean.NavigationC"/>
2. <bean id="b" class="cn.javass.spring.chapter3.bean.NavigationB">
3. <property name="list"><list></list></property>
4. <property name="map"><map></map></property>
5. <property name="properties"><props></props></property>
6. </bean>
```

配置完需要被导航的Bean定义了，该来配置NavigationA导航Bean了，在此需要注意，由于“navigationB”属性为空值，在此需要首先注入“navigationB”值；还有对于数组导航不能越界否则报错；具体配置如下：

```
1. <bean id="a" class="cn.javass.spring.chapter3.bean.NavigationA">
2. <!-- 首先注入navigatiionB 确保它非空 -->
3. <property name="navigationB" ref="b"/>
4. <!-- 对象图导航注入 -->
5. <property name="navigationB.navigationC" ref="c"/>
6. <!-- 注入列表数据类型数据 -->
7. <property name="navigationB.list[0]" ref="c"/>
8. <!-- 注入map类型数据 -->
9. <property name="navigationB.map[key]" ref="c"/>
10. <!-- 注入properties类型数据 -->
11. <property name="navigationB.properties[0]" ref="c"/>
12. <!-- 注入properties类型数据 -->
13. <property name="navigationB.properties[1]" ref="c"/>
14. <!-- 注入数组类型数据 ，注意不要越界-->
```

```
15. <property name="navigationB.array[0]" ref="c"/>
16. </bean>
```

配置完毕，具体测试代码在cn.javass.spring.chapter3. DependencyInjectTest，让我们看下测试代码吧：

```
1. //对象图导航
2. @Test
3. public void testNavigationBeanInject() {
4. ApplicationContext context =
5. new ClassPathXmlApplicationContext("chapter3/navigationBeanInject.xml");
6. NavigationA navigationA = context.getBean("a", NavigationA.class);
7. navigationA.getNavigationB().getNavigationC().sayNavigation();
8. navigationA.getNavigationB().getList().get(0).sayNavigation();
9. navigationA.getNavigationB().getMap().get("key").sayNavigation();
10. navigationA.getNavigationB().getArray()[0].sayNavigation();
11. ((NavigationC)navigationA.getNavigationB().getProperties().get("1"))
12. .sayNavigation();
13. }
```

测试完毕，应该输出5个“===navigation c”，是不是很简单，注意这种方式是不推荐使用的，了解一下就够了，最好使用3.1.5一节使用的配置方式。

3.1.11配置简写

让我们来总结一下依赖注入配置及简写形式，其实我们已经在以上部分穿插着进行简化配置了：

一、构造器注入：

1）常量值

简写：<constructor-arg index="0" value="常量"/>

全写：<constructor-arg index="0"><value>常量</value></constructor-arg>

2）引用

简写：<constructor-arg index="0" ref="引用"/>

全写：<constructor-arg index="0"><ref bean="引用"/></constructor-arg>

二、**setter**注入：

1）常量值

简写：<property name="message" value="常量"/>

全写：<property name="message"><value>常量</value></ property>

2）引用

简写：<property name="message" ref="引用"/>

全写：<property name="message"><ref bean="引用"/></ property>

3）数组：<array>没有简写形式

4）列表：<list>没有简写形式

5）集合：<set>没有简写形式

6）字典

简写：<map>

<entry key="键常量" value="值常量"/>

<entry key-ref="键引用" value-ref="值引用"/>

</map>

全写：<map>

<entry><key><value>键常量</value></key><value>值常量</value></entry>

<entry><key><ref bean="键引用"/></key><ref bean="值引用"/></entry>

</map>

7）Properties：没有简写形式

三、使用**p**命名空间简化**setter**注入：

使用p命名空间来简化setter注入，具体使用如下：

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5. xsi:schemaLocation="
6. http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"&gt;
8. <bean id="bean1" class="java.lang.String">
9. <constructor-arg index="0" value="test"/>
10. </bean>
11. <bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
```

12. p:id="value"/>
13. <bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
14. p:id-ref="bean1"/>
15. </beans>

16. **xmlns:p="http://www.springframework.org/schema/p"** ：首先指定**p**命名空间；

17. **<bean id="……" class="……" p:id="value"/>** ：常量setter注入方式，其等价于<property name="id" value="value"/>；
    -
        - **<bean id="……" class="……" p:id-ref="bean1"/>** ：引用setter注入方式，其等价于<property name="id" ref="bean1"/>。

原创内容，转载请注明【http://sishuok.com/forum/posts/list/2447.html】

【第三章】 DI 之 3.2 循环依赖 ——跟我学spring3

3.2.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个Bean相互之间的持有对方，比如CircleA引用CircleB，CircleB引用CircleC，CircleC引用CircleA，则它们最终反映为一个环。此处不是循环调用，循环调用是方法之间的环调用。如图3-5所示：

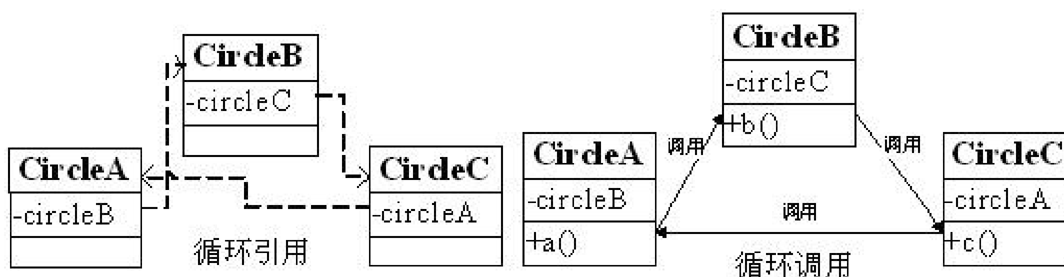


图3-5 循环引用

循环调用是无法解决的，除非有终结条件，否则就是死循环，最终导致内存溢出错误。

Spring容器循环依赖包括构造器循环依赖和setter循环依赖，那Spring容器如何解决循环依赖呢？首先让我们来定义循环引用类：

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class CircleA {
3.  private CircleB circleB;
4.  public CircleA() {
5.  }
6.  public CircleA(CircleB circleB) {
7.  this.circleB = circleB;
8.  }
9.  public void setCircleB(CircleB circleB)
10. {
11. this.circleB = circleB;
12. }
13. public void a() {
14. circleB.b();
15. }
16. }
```

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class CircleB {
```

```
3. private CircleC circleC;
4. public CircleB() {
5. }
6. public CircleB(CircleC circleC) {
7. this.circleC = circleC;
8. }
9. public void setCircleC(CircleC circleC)
10. {
11. this.circleC = circleC;
12. }
13. public void b() {
14. circleC.c();
15. }
16. }
```

```
1. package cn.javass.spring.chapter3.bean;
2. public class CircleC {
3. private CircleA circleA;
4. public CircleC() {
5. }
6. public CircleC(CircleA circleA) {
7. this.circleA = circleA;
8. }
9. public void setCircleA(CircleA circleA)
10. {
11. this.circleA = circleA;
12. }
13. public void c() {
14. circleA.a();
15. }
16. }
```

3.2.2 Spring如何解决循环依赖

一、构造器循环依赖：表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException异常表示循环依赖。

如在创建CircleA类时，构造器需要CircleB类，那将去创建CircleB，在创建CircleB类时又发现需要CircleC类，则又去创建CircleC，最终在创建CircleC时发现又需要CircleA；从而形成一个环，没办法创建。

Spring容器将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

1）首先让我们看一下配置文件（chapter3/circleInjectByConstructor.xml）：

```
1. <bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA">
2. <constructor-arg index="0" ref="circleB"/>
3. </bean>
4. <bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB">
5. <constructor-arg index="0" ref="circleC"/>
6. </bean>
7. <bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC">
8. <constructor-arg index="0" ref="circleA"/>
9. </bean>
```

2）写段测试代码（cn.javass.spring.chapter3.CircleTest）测试一下吧：

```
1. @Test(expected = BeanCurrentlyInCreationException.class)
2. public void testCircleByConstructor() throws Throwable {
3. try {
4. new ClassPathXmlApplicationContext("chapter3/circleInjectByConstructor.xml");
5. }
6. catch (Exception e) {
7. //因为要在创建circle3时抛出；
8. Throwable e1 = e.getCause().getCause().getCause();
9. throw e1;
10. }
11. }
```

让我们分析一下吧：

1、Spring容器创建“circleA” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleB”，并将“circleA” 标识符放到“当前创建Bean池”；

2、Spring容器创建“circleB” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleC”，并将“circleB” 标识符放到“当前创建Bean池”；

3、Spring容器创建“circleC” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleA”，并将“circleC” 标识符放到“当前创建Bean池”；

4、到此为止Spring容器要去创建“circleA”Bean，发现该Bean 标识符在“当前创建Bean池”中，因为表示循环依赖，抛出BeanCurrentlyInCreationException。

二、**setter**循环依赖：表示通过setter注入方式构成的循环依赖。

对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入）的Bean来完成的，而且只能解决单例作用域的Bean循环依赖。

如下代码所示，通过提前暴露一个单例工厂方法，从而使其他Bean能引用到该Bean。

```
1. addSingletonFactory(beanName, new ObjectFactory() {
2. public Object getObject() throws BeansException {
3. return getEarlyBeanReference(beanName, mbd, bean);
4. }
5. });
```

具体步骤如下：

1、Spring容器创建单例“circleA” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory ”用于返回一个提前暴露一个创建中的Bean，并将“circleA” 标识符放到“当前创建Bean池”；然后进行setter注入“circleB”；

2、Spring容器创建单例“circleB” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleB” 标识符放到“当前创建Bean池”，然后进行setter注入“circleC”；

3、Spring容器创建单例“circleC” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory ”用于返回一个提前暴露一个创建中的Bean，并将“circleC” 标识符放到“当前创建Bean池”，然后进行setter注入“circleA”；进行注入“circleA”时由于提前暴露了“ObjectFactory”工厂从而使用它返回提前暴露一个创建中的Bean；

4、最后在依赖注入“circleB”和“circleA”，完成setter注入。

对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。

```
1. <!-- 定义Bean配置文件，注意scope都是“prototype”-->
2. <bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA" scope="prototype">
3. <property name="circleB" ref="circleB"/>
4. </bean>
5. <bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB" scope="prototype">
6. <property name="circleC" ref="circleC"/>
7. </bean>
8. <bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC" scope="prototype">
9. <property name="circleA" ref="circleA"/>
```

```
10. </bean>
```

```
1. //测试代码cn.javass.spring.chapter3.CircleTest
2. @Test(expected = BeanCurrentlyInCreationException.class)
3. public void testCircleBySetterAndPrototype () throws Throwable {
4. try {
5. ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
6. "chapter3/circleInjectBySetterAndPrototype.xml");
7. System.out.println(ctx.getBean("circleA"));
8. }
9. catch (Exception e) {
10. Throwable e1 = e.getCause().getCause().getCause();
11. throw e1;
12. }
13. }
```

对于“singleton”作用域Bean，可以通过“setAllowCircularReferences(false);”来禁用循环引用：

```
1. @Test(expected = BeanCurrentlyInCreationException.class)
2. public void testCircleBySetterAndSingleton2() throws Throwable {
3. try {
4. ClassPathXmlApplicationContext ctx =
5. new ClassPathXmlApplicationContext();
6. ctx.setConfigLocation("chapter3/circleInjectBySetterAndSingleton.xml");
7. ctx.refresh();
8. }
9. catch (Exception e) {
10. Throwable e1 = e.getCause().getCause().getCause();
11. throw e1;
12. }
13. }
```

补充：出现循环依赖是设计上的问题，一定要避免！

请参考《敏捷软件开发：原则、模式与实践》中的“无环依赖”原则

包之间的依赖结构必须是一个直接的无环图形（DAG）。也就是说，在依赖结构中不允许出现环（循环依赖）。

原创内容 转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2448.html#7070】

【第三章】 DI 之 3.3 更多DI的知识 ——跟我学spring3

3.3.1 延迟初始化Bean

延迟初始化也叫做惰性初始化，指不提前初始化Bean，而是只有在真正使用时才创建及初始化Bean。

配置方式很简单只需在<bean>标签上指定 “lazy-init” 属性值为“true”即可延迟初始化Bean。

Spring容器会在创建容器时提前初始化“singleton”作用域的Bean，“singleton”就是单例的意思即整个容器每个Bean只有一个实例，后边会详细介绍。Spring容器预先初始化Bean通常能帮助我们提前发现配置错误，所以如果没有什么情况建议开启，除非有某个Bean可能需要加载很大资源，而且很可能在整个应用程序生命周期中很可能使用不到，可以设置为延迟初始化。

延迟初始化的Bean通常会在第一次使用时被初始化；或者在被非延迟初始化Bean作为依赖对象注入时在会随着初始化该Bean时被初始化，因为在这时使用了延迟初始化Bean。

容器管理初始化Bean消除了编程实现延迟初始化，完全由容器控制，只需在需要延迟初始化的Bean定义上配置即可，比编程方式更简单，而且是无侵入代码的。

具体配置如下：

```
1. <bean id="helloApi"
2. class="cn.javass.spring.chapter2.helloworld.HelloImpl"
3. lazy-init="true"/>
```

3.3.2 使用depends-on

depends-on是指指定Bean初始化及销毁时的顺序，使用depends-on属性指定的Bean要先初始化完毕后才初始化当前Bean，由于只有“singleton”Bean能被Spring管理销毁，所以当指定的Bean都是“singleton”时，使用depends-on属性指定的Bean要在指定的Bean之后销毁。

配置方式如下：

```
1. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <bean id="decorator"
3. class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
4. depends-on="helloApi">
5. <property name="helloApi"><ref bean="helloApi"/></property>
6. </bean>
```

“decorator”指定了“depends-on”属性为“helloApi”，所以在“decorator”Bean初始化之前要先初始化“helloApi”，而在销毁“helloApi”之前先要销毁“decorator”，大家注意一下销毁顺序，与文档上的不符。

“depends-on”属性可以指定多个Bean，若指定多个Bean可以用“;”、“,”、空格分割。

那“depends-on”有什么好处呢？主要是给出明确的初始化及销毁顺序，比如要初始化“decorator”时要确保“helloApi”Bean的资源准备好了，否则使用“decorator”时会看不到准备的资源；而在销毁时要先在“decorator”Bean的把对“helloApi”资源的引用释放掉才能销毁“helloApi”，否则可能销毁 “helloApi”时而“decorator”还保持着资源访问，造成资源不能释放或释放错误。

让我们看个例子吧，在平常开发中我们可能需要访问文件系统，而文件打开、关闭是必须配对的，不能打开后不关闭，从而造成其他程序不能访问该文件。让我们来看具体配置吧：

1）准备测试类：

ResourceBean从配置文件中配置文件位置，然后定义初始化方法init中打开指定的文件，然后获取文件流；最后定义销毁方法destroy用于在应用程序关闭时调用该方法关闭掉文件流。

DependentBean中会注入ResourceBean，并从ResourceBean中获取文件流写入内容；定义初始化方法init用来定义一些初始化操作并向文件中输出文件头信息；最后定义销毁方法用于在关闭应用程序时想文件中输出文件尾信息。

具体代码如下：

```
1.  package cn.javass.spring.chapter3.bean;
2.  import java.io.File;
3.  import java.io.FileNotFoundException;
4.  import java.io.FileOutputStream;
5.  import java.io.IOException;
6.  public class ResourceBean {
7.  private FileOutputStream fos;
8.  private File file;
9.  //初始化方法
10. public void init() {
11. System.out.println("ResourceBean:========初始化");
12. //加载资源,在此只是演示
13. System.out.println("ResourceBean:========加载资源，执行一些预操作");
14. try {
15. this.fos = new FileOutputStream(file);
16. } catch (FileNotFoundException e) {
17. e.printStackTrace();
18. }
```

```
19. }
20. //销毁资源方法
21. public void destroy() {
22. System.out.println("ResourceBean:========销毁");
23. //释放资源
24. System.out.println("ResourceBean:========释放资源，执行一些清理操作");
25. try {
26. fos.close();
27. } catch (IOException e) {
28. e.printStackTrace();
29. }
30. }
31. public FileOutputStream getFos() {
32. return fos;
33. }
34. public void setFile(File file) {
35. this.file = file;
36. }
37. }
```

```
1. package cn.javass.spring.chapter3.bean;
2. import java.io.IOException;
3. public class DependentBean {
4. ResourceBean resourceBean;
5. public void write(String ss) throws IOException {
6. System.out.println("DependentBean:=======写资源");
7. resourceBean.getFos().write(ss.getBytes());
8. }
9. //初始化方法
10. public void init() throws IOException {
11. System.out.println("DependentBean:=======初始化");
12. resourceBean.getFos().write("DependentBean:=======初始化=====".getBytes());
13. }
14. //销毁方法
15. public void destroy() throws IOException {
16. System.out.println("DependentBean:=======销毁");
17. //在销毁之前需要往文件中写销毁内容
18. resourceBean.getFos().write("DependentBean:=======销毁=====".getBytes());
19. }
20. public void setResourceBean(ResourceBean resourceBean) {
```

```
21. this.resourceBean = resourceBean;
22. }
23. }
```

2）类定义好了，让我们来进行Bean定义吧，具体配置文件如下：

```
1. <bean id="resourceBean"
2. class="cn.javass.spring.chapter3.bean.ResourceBean"
3. init-method="init" destroy-method="destroy">
4. <property name="file" value="D:/test.txt"/>
5. </bean>
6. <bean id="dependentBean"
7. class="cn.javass.spring.chapter3.bean.DependentBean"
8. init-method="init" destroy-method="destroy" depends-on="resourceBean">
9. <property name="resourceBean" ref="resourceBean"/>
10. </bean>
```

<property name="file" value="D:/test.txt"/>配置：Spring容器能自动把字符串转换为java.io.File。

init-method="init" ：指定初始化方法，在构造器注入和setter注入完毕后执行。

destroy-method="destroy"：指定销毁方法，只有“singleton”作用域能销毁，“prototype”作用域的一定不能，其他作用域不一定能；后边再介绍。

在此配置中，resourceBean初始化在dependentBean之前被初始化，resourceBean销毁会在dependentBean销毁之后执行。

3）配置完毕，测试一下吧：

```
1. package cn.javass.spring.chapter3;
2. import java.io.IOException;
3. import org.junit.Test;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import cn.javass.spring.chapter3.bean.DependentBean;
6. public class MoreDependencyInjectTest {
7. @Test
8. public void testDependOn() throws IOException {
9. ClassPathXmlApplicationContext context =
10. new ClassPathXmlApplicationContext("chapter3/depends-on.xml");
11. //一点要注册销毁回调，否则我们定义的销毁方法不执行
12. context.registerShutdownHook();
13. DependentBean dependentBean =
14. context.getBean("dependentBean", DependentBean.class);
```

```
15. dependentBean.write("aaa");
16. }
17. }
```

测试跟其他测试完全一样，只是在此我们一定要注册销毁方法回调，否则销毁方法不会执行。

如果配置没问题会有如下输出：

```
1. ResourceBean:========初始化
2. ResourceBean:========加载资源，执行一些预操作
3. DependentBean:=========初始化
4. DependentBean:=========写资源
5. DependentBean:=========销毁
6. ResourceBean:========销毁
7. ResourceBean:========释放资源，执行一些清理操作
```

3.3.3 自动装配

自动装配就是指由Spring来自动地注入依赖对象，无需人工参与。

目前Spring3.0支持“no”、“byName ”、“byType”、“constructor”四种自动装配，默认是“no”指不支持自动装配的，其中Spring3.0已不推荐使用之前版本的“autodetect”自动装配，推荐使用Java 5+支持的（@Autowired）注解方式代替；如果想支持“autodetect”自动装配，请将schema改为“spring-beans-2.5.xsd”或去掉。

自动装配的好处是减少构造器注入和setter注入配置，减少配置文件的长度。自动装配通过配置<bean>标签的“autowire”属性来改变自动装配方式。接下来让我们挨着看下配置的含义。

一、**default**：表示使用默认的自动装配，默认的自动装配需要在<beans>标签中使用default-autowire属性指定，其支持“no”、“byName ”、“byType”、“constructor”四种自动装配，如果需要覆盖默认自动装配，请继续往下看；

二、**no**：意思是不支持自动装配，必须明确指定依赖。

三、**byName**：通过设置Bean定义属性autowire="byName"，意思是根据名字进行自动装配，只能用于setter注入。比如我们有方法“setHelloApi”，则“byName”方式Spring容器将查找名字为helloApi的Bean并注入，如果找不到指定的Bean，将什么也不注入。

例如如下Bean定义配置：

```
1. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
3. autowire="byName"/>
```

测试代码如下：

```
1. package cn.javass.spring.chapter3;
2. import java.io.IOException;
3. import org.junit.Test;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import cn.javass.spring.chapter2.helloworld.HelloApi;
6. public class AutowireBeanTest {
7. @Test
8. public void testAutowireByName() throws IOException {
9. ClassPathXmlApplicationContext context =
10. new ClassPathXmlApplicationContext("chapter3/autowire-byName.xml");
11. HelloApi helloApi = context.getBean("bean", HelloApi.class);
12. helloApi.sayHello();
13. }
14. }
```

是不是不要配置<property>了，如果一个bean有很多setter注入，通过“byName”方式是不是能减少很多<property>配置。此处注意了，在根据名字注入时，将把当前**Bean**自己排除在外：比如“hello”Bean类定义了“setHello”方法，则hello是不能注入到“setHello”的。

四、**“byType”**：通过设置Bean定义属性autowire="byType"，意思是指根据类型注入，用于setter注入，比如如果指定自动装配方式为“byType”，而“setHelloApi”方法需要注入HelloApi类型数据，则Spring容器将查找HelloApi类型数据，如果找到一个则注入该Bean，如果找不到将什么也不注入，如果找到多个Bean将优先注入<bean>标签“primary”属性为true的Bean，否则抛出异常来表明有个多个Bean发现但不知道使用哪个。让我们用例子来讲解一下这几种情况吧。

1）根据类型只找到一个Bean，此处注意了，在根据类型注入时，将把当前**Bean**自己排除在外，即如下配置中helloApi和bean都是HelloApi接口的实现，而“bean”通过类型进行注入“HelloApi”类型数据时自己是排除在外的，配置如下（具体测试请参考AutowireBeanTest.testAutowireByType1方法）：

```
1. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
3. autowire="byType"/>
```

2）根据类型找到多个Bean时，对于集合类型（如List、Set）将注入所有匹配的候选者，而对于其他类型遇到这种情况可能需要使用“autowire-candidate”属性为false来让指定的Bean放弃作为自动装配的候选者，或使用“primary”属性为true来指定某个Bean为首选Bean：

2.1）通过设置Bean定义的“autowire-candidate”属性为false来把指定Bean后自动装配候选者中移除：

```
1. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <!-- 从自动装配候选者中去除 -->
```

```
3. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"
4. autowire-candidate="false"/>
5. <bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
6. autowire="byType"/>
```

2.2）通过设置Bean定义的“primary”属性为true来把指定自动装配时候选者中首选Bean：

```
1. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <!-- 自动装配候选者中的首选Bean-->
3. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
4. <bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
5. autowire="byType"/>
```

具体测试请参考AutowireBeanTest类的testAutowireByType*方法。

五、**“constructor”**：通过设置Bean定义属性autowire="constructor"，功能和“byType”功能一样，根据类型注入构造器参数，只是用于构造器注入方式，直接看例子吧：

```
1. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <!-- 自动装配候选者中的首选Bean-->
3. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
4. <bean id="bean"
5. class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
6. autowire="constructor"/>
```

测试代码如下：

```
1. @Test
2. public void testAutowireByConstructor() throws IOException {
3. ClassPathXmlApplicationContext context =
4. new ClassPathXmlApplicationContext("chapter3/autowire-byConstructor.xml");
5. HelloApi helloApi = context.getBean("bean", HelloApi.class);
6. helloApi.sayHello();
7. }
```

六、**autodetect**：自动检测是使用“constructor”还是“byType”自动装配方式，已不推荐使用。如果Bean有空构造器那么将采用“byType”自动装配方式，否则使用“constructor”自动装配方式。此处要把3.0的xsd替换为2.5的xsd，否则会报错。

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xsi:schemaLocation="
```

```
6.  http://www.springframework.org/schema/beans
7.  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8.  http://www.springframework.org/schema/context
9.  http://www.springframework.org/schema/context/spring-context-2.5.xsd"&gt;
10. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
11. <!-- 自动装配候选者中的首选Bean-->
12. <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
13. <bean id="bean"
14. class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
15. autowire="autodetect"/>
16. </beans>
```

可以采用在“<beans>”标签中通过“default-autowire”属性指定全局的自动装配方式，即如果default-autowire=”byName”，将对所有Bean进行根据名字进行自动装配。

不是所有类型都能自动装配：

- 不能自动装配的数据类型：Object、基本数据类型（Date、CharSequence、Number、URI、URL、Class、int）等；
- 通过“<beans>”标签default-autowire-candidates属性指定的匹配模式，不匹配的将不能作为自动装配的候选者，例如指定“*Service*，Dao”，将只把匹配这些模式的Bean作为候选者，而不匹配的不会作为候选者；
- 通过将“<bean>”标签的autowire-candidate属性可被设为false，从而该Bean将不会作为依赖注入的候选者。

数组、集合、字典类型的根据类型自动装配和普通类型的自动装配是有区别的：

- 数组类型、集合（**Set**、**Collection**、**List**）接口类型：将根据泛型获取匹配的所有候选者并注入到数组或集合中，如“List<HelloApi> list”将选择所有的HelloApi类型Bean并注入到list中，而对于集合的具体类型将只选择一个候选者，“如 ArrayList<HelloApi> list”将选择一个类型为ArrayList的Bean注入，而不是选择所有的HelloApi类型Bean进行注入；
- 字典（**Map**）接口类型：同样根据泛型信息注入，键必须为String类型的Bean名字，值根据泛型信息获取，如“Map<String, HelloApi> map” 将选择所有的HelloApi类型Bean并注入到map中，而对于具体字典类型如“HashMap<String, HelloApi> map”将只选择类型为HashMap的Bean注入，而不是选择所有的HelloApi类型Bean进行注入。

自动装配我们已经介绍完了，自动装配能带给我们什么好处呢？首先，自动装配确实减少了配置文件的量；其次， “byType”自动装配能在相应的Bean更改了字段类型时自动更新，即修改Bean类不需要修改配置，确实简单了。

自动装配也是有缺点的，最重要的缺点就是没有了配置，在查找注入错误时非常麻烦，还有比如基本类型没法完成自动装配，所以可能经常发生一些莫名其妙的错误，在此我推荐大家不要使用该方式，最好是指定明确的注入方式，或者采用最新的Java5+注解注入方式。所以大家在使用自动装配时应该考虑自己负责项目的复杂度来进行衡量是否选择自动装配方式。

自动装配注入方式能和配置注入方式一同工作吗？当然可以，大家只需记住配置注入的数据会覆盖自动装配注入的数据。

大家是否注意到对于采用自动装配方式时如果没找到合适的的Bean时什么也不做，这样在程序中总会莫名其妙的发生一些空指针异常，而且是在程序运行期间才能发现，有没有办法能在提前发现这些错误呢？接下来就让我来看下依赖检查吧。

3.3.4 依赖检查

上一节介绍的自动装配，很可能发生没有匹配的Bean进行自动装配，如果此种情况发生，只有在程序运行过程中发生了空指针异常才能发现错误，如果能提前发现该多好啊，这就是依赖检查的作用。

依赖检查：用于检查Bean定义的属性都注入数据了，不管是自动装配的还是配置方式注入的都能检查，如果没有注入数据将报错，从而提前发现注入错误，只检查具有setter方法的属性。

Spring3+也不推荐配置方式依赖检查了，建议采用Java5+ @Required注解方式，测试时请将XML schema降低为2.5版本的，和自动装配中“autodetect”配置方式的xsd一样。

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="
5. http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
7. </beans>
```

依赖检查有none、simple、object、all四种方式，接下来让我们详细介绍一下：

一、**none**：默认方式，表示不检查；

二、**objects**：检查除基本类型外的依赖对象，配置方式为：dependency-check="objects"，此处我们为HelloApiDecorator添加一个String类型属性“message”，来测试如果有简单数据类型的属性为null，也不报错；

```
1. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <!-- 注意我们没有注入helloApi，所以测试时会报错 -->
3. <bean id="bean"
4. class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
5. dependency-check="objects">
6. <property name="message" value="Haha"/>
7. </bean>
```

注意由于我们没有注入bean需要的依赖“helloApi”，所以应该抛出异常UnsatisfiedDependencyException，表示没有发现满足的依赖：

```
1. package cn.javass.spring.chapter3;
2. import java.io.IOException;
3. import org.junit.Test;
4. import org.springframework.beans.factory.UnsatisfiedDependencyException;
5. import org.springframework.context.support.ClassPathXmlApplicationContext;
6. public class DependencyCheckTest {
7. @Test(expected = UnsatisfiedDependencyException.class)
8. public void testDependencyCheckByObject() throws IOException {
9. //将抛出异常
10. new ClassPathXmlApplicationContext("chapter3/dependency-check-object.xml");
11. }
12. }
```

三、**simple**：对基本类型进行依赖检查，包括数组类型，其他依赖不报错；配置方式为：dependency-check="simple"，以下配置中没有注入message属性，所以会抛出异常：

```
1. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <!-- 注意我们没有注入message属性，所以测试时会报错 -->
3. <bean id="bean"
4. class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
5. dependency-check="simple">
6. <property name="helloApi" ref="helloApi"/>
7. </bean>
```

四、**all**：对所以类型进行依赖检查，配置方式为：dependency-check="all"，如下配置方式中如果两个属性其中一个没配置将报错。

```
1. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <bean id="bean"
3. class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
4. dependency-check="all">
5. <property name="helloApi" ref="helloApi"/>
6. <property name="message" value="Haha"/>
7. </bean>
```

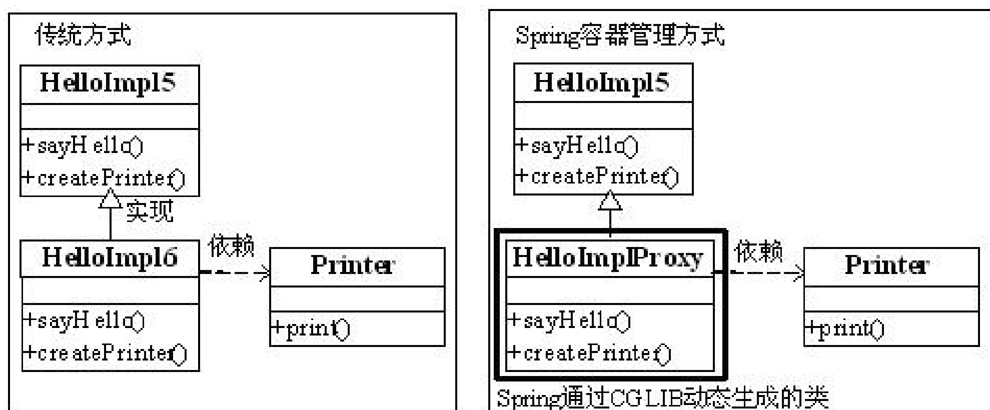
依赖检查也可以通过“<beans>”标签中default-dependency-check属性来指定全局依赖检查配置。

3.3.5 方法注入

所谓方法注入其实就是通过配置方式覆盖或拦截指定的方法，通常通过代理模式实现。Spring提供两种方法注入：查找方法注入和方法替换注入。

因为Spring是通过CGLIB动态代理方式实现方法注入，也就是通过动态修改类的字节码来实现的，本质就是生成需方法注入的类的子类方式实现。

在进行测试之前，我们需要确保将“com.springsource.cn.sf.cglib-2.2.0.jar”放到lib里并添加到“Java Build Path”中的Libararies中。否则报错，异常中包含**“nested exception is java.lang.NoClassDefFoundError: cn/sf/cglib/proxy/CallbackFilter”**。



传统方式和Spring容器管理方式唯一不同的是不需要我们手动生成子类，而是通过配置方式来实现；其中如果要替换createPrinter()方法的返回值就使用查找方法注入；如果想完全替换sayHello()方法体就使用方法替换注入。 接下来让我们看看具体实现吧。

一、查找方法注入：又称为Lookup方法注入，用于注入方法返回结果，也就是说能通过配置方式替换方法返回结果。使用<lookup-method name="方法名" bean="bean名字"/>配置；其中name属性指定方法名，bean属性指定方法需返回的Bean。

方法定义格式：访问级别必须是public或protected，保证能被子类重载，可以是抽象方法，必须有返回值，必须是无参数方法，查找方法的类和被重载的方法必须为非final：

<public|protected> [abstract] <return-type> theMethodName(no-arguments);

因为“singleton”Bean在容器中只有一个实例，而“prototype”Bean是每次获取容器都返回一个全新的实例，所以如果“singleton”Bean在使用“prototype” Bean情况时，那么“prototype”Bean由于是“singleton”Bean的一个字段属性，所以获取的这个“prototype”Bean就和它所在的“singleton”Bean具有同样的生命周期，所以不是我们所期待的结果。因此查找方法注入就是用于解决这个问题。

1） 首先定义我们需要的类，Printer类是一个有状态的类，counter字段记录访问次数：

```
1. package cn.javass.spring.chapter3.bean;
2. public class Printer {
3. private int counter = 0;
4. public void print(String type) {
```

```
5. System.out.println(type + " printer: " + counter++);
6. }
7. }
```

HelloImpl5类用于打印欢迎信息，其中包括setter注入和方法注入，此处特别需要注意的是该类是抽象的，充分说明了需要容器对其进行子类化处理，还定义了一个抽象方法createPrototypePrinter用于创建“prototype”Bean，createSingletonPrinter方法用于创建“singleton”Bean，此处注意方法会被Spring拦截，不会执行方法体代码：

```
1. package cn.javass.spring.chapter3;
2. import cn.javass.spring.chapter2.helloworld.HelloApi;
3. import cn.javass.spring.chapter3.bean.Printer;
4. public abstract class HelloImpl5 implements HelloApi {
5. private Printer printer;
6. public void sayHello() {
7. printer.print("setter");
8. createPrototypePrinter().print("prototype");
9. createSingletonPrinter().print("singleton");
10. }
11. public abstract Printer createPrototypePrinter();
12. public Printer createSingletonPrinter() {
13. System.out.println("该方法不会被执行，如果输出就错了");
14. return new Printer();
15. }
16. public void setPrinter(Printer printer) {
17. this.printer = printer;
18. }
19. }
```

2） 开始配置了，配置文件在（resources/chapter3/lookupMethodInject.xml），其中“prototypePrinter”是“prototype”Printer，“singletonPrinter”是“singleton”Printer，“helloApi1”是“singleton”Bean，而“helloApi2”注入了“prototype”Bean：

```
1. <bean id="prototypePrinter"
2. class="cn.javass.spring.chapter3.bean.Printer" scope="prototype"/>
3. <bean id="singletonPrinter"
4. class="cn.javass.spring.chapter3.bean.Printer" scope="singleton"/>
5. <bean id="helloApi1" class="cn.javass.spring.chapter3.HelloImpl5" scope="singleton">
6. <property name="printer" ref="prototypePrinter"/>
7. <lookup-method name="createPrototypePrinter" bean="prototypePrinter"/>
8. <lookup-method name="createSingletonPrinter" bean="singletonPrinter"/>
9. </bean>
```

```
10. <bean id="helloApi2" class="cn.javass.spring.chapter3.HelloImpl5" scope="prototype">
11. <property name="printer" ref="prototypePrinter"/>
12. <lookup-method name="createPrototypePrinter" bean="prototypePrinter"/>
13. <lookup-method name="createSingletonPrinter" bean="singletonPrinter"/>
14. </bean>
```

3）测试代码如下：

```
1. package cn.javass.spring.chapter3;
2. import org.junit.Test;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import cn.javass.spring.chapter2.helloworld.HelloApi;
5. public class MethodInjectTest {
6. @Test
7. public void testLookup() {
8. ClassPathXmlApplicationContext context =
9. new ClassPathXmlApplicationContext("chapter3/lookupMethodInject.xml");
10. System.out.println("=======singleton sayHello======");
11. HelloApi helloApi1 = context.getBean("helloApi1", HelloApi.class);
12. helloApi1.sayHello();
13. helloApi1 = context.getBean("helloApi1", HelloApi.class);
14. helloApi1.sayHello();
15. System.out.println("=======prototype sayHello======");
16. HelloApi helloApi2 = context.getBean("helloApi2", HelloApi.class);
17. helloApi2.sayHello();
18. helloApi2 = context.getBean("helloApi2", HelloApi.class);
19. helloApi2.sayHello();
20. }}
```

其中“helloApi1”测试中，其输出结果如下：

```
1. =======singleton sayHello======
2. setter printer: 0
3. prototype printer: 0
4. singleton printer: 0
5. setter printer: 1
6. prototype printer: 0
7. singleton printer: 1
```

首先“helloApi1”是“singleton”，通过setter注入的“printer”是“prototypePrinter”，所以它应该输出“setter printer:0”和“setter printer:1”；而“createPrototypePrinter”方法注入了“prototypePrinter”，所以应该输出两次“prototype printer:0”；而“createSingletonPrinter”注入了“singletonPrinter”，所以应该输出“singleton printer:0”和“singleton printer:1”。

而“helloApi2”测试中，其输出结果如下：

```
1. =======prototype sayHello======
2. setter printer: 0
3. prototype printer: 0
4. singleton printer: 2
5. setter printer: 0
6. prototype printer: 0
7. singleton printer: 3
```

首先“helloApi2”是“prototype”，通过setter注入的“printer”是“prototypePrinter”，所以它应该输出两次“setter printer:0”；而“createPrototypePrinter”方法注入了“prototypePrinter”，所以应该输出两次“prototype printer:0”；而“createSingletonPrinter”注入了“singletonPrinter”，所以应该输出“singleton printer:2”和“singleton printer:3”。

大家是否注意到“createSingletonPrinter”方法应该输出“该方法不会被执行，如果输出就错了”，而实际是没输出的，这说明Spring拦截了该方法并使用注入的Bean替换了返回结果。

方法注入主要用于处理“singleton”作用域的Bean需要其他作用域的Bean时，采用Spring查找方法注入方式无需修改任何代码即能获取需要的其他作用域的Bean。

二、替换方法注入：也叫“MethodReplacer”注入，和查找注入方法不一样的是，他主要用来替换方法体。通过首先定义一个MethodReplacer接口实现，然后如下配置来实现：

```
1. <replaced-method name="方法名" replacer="MethodReplacer实现">
2. <arg-type>参数类型</arg-type>
3. </replaced-method>”
```

1）首先定义MethodReplacer实现，完全替换掉被替换方法的方法体及返回值，其中reimplement方法重定义方法 功能，参数obj为被替换方法的对象，method为被替换方法，args为方法参数；最需要注意的是不能再 通过“method.invoke(obj, new String[]{"hehe"});” 反射形式再去调用原来方法，这样会产生循环调用；如果返回值类型为Void，请在实现中返回null：

```
1. package cn.javass.spring.chapter3.bean;
2. import java.lang.reflect.Method;
3. import org.springframework.beans.factory.support.MethodReplacer;
4. public class PrinterReplacer implements MethodReplacer {
5. @Override
6. public Object reimplement(Object obj, Method method, Object[] args) throws Throwable
   {
7. System.out.println("Print Replacer");
8. //注意此处不能再通过反射调用了,否则会产生循环调用，知道内存溢出
9. //method.invoke(obj, new String[]{"hehe"});
```

```
10. return null;
11. }
12. }
```

2）配置如下，首先定义MethodReplacer实现，使用< replaced-method >标签来指定要进行替换方法，属性name指定替换的方法名字，replacer指定该方法的重新实现者，子标签< arg-type >用来指定原来方法参数的类型，必须指定否则找不到原方法：

```
1. <bean id="replacer" class="cn.javass.spring.chapter3.bean.PrinterReplacer"/>
2. <bean id="printer" class="cn.javass.spring.chapter3.bean.Printer">
3. <replaced-method name="print" replacer="replacer">
4. <arg-type>java.lang.String</arg-type>
5. </replaced-method>
6. </bean>
```

3）测试代码将输出“Print Replacer ”，说明方法体确实被替换了：

```
1. @Test
2. public void testMethodReplacer() {
3. ClassPathXmlApplicationContext context = new
   ClassPathXmlApplicationContext("chapter3/methodReplacerInject.xml");
4. Printer printer = context.getBean("printer", Printer.class);
5. printer.print("我将被替换");
6. }
```

【第三章】 DI 之 3.4 Bean的作用域 ——跟我学spring3

3.4 Bean的作用域

什么是作用域呢？即“scope”，在面向对象程序设计中一般指对象或变量之间的可见范围。而在Spring容器中是指其创建的Bean对象相对于其他Bean对象的请求可见范围。

Spring提供“singleton”和“prototype”两种基本作用域，另外提供“request”、“session”、“global session”三种web作用域；Spring还允许用户定制自己的作用域。

3.4.1 基本的作用域

一、**singleton**：指“singleton”作用域的Bean只会在每个Spring IoC容器中存在一个实例，而且其完整生命周期完全由Spring容器管理。对于所有获取该Bean的操作Spring容器将只返回同一个Bean。

> |
>
> GoF单例设计模式指“保证一个类仅有一个实例，并提供一个访问它的全局访问点”，介绍了两种实现：通过在类上定义静态属性保持该实例和通过注册表方式。
>
> |

1）通过在类上定义静态属性保持该实例：一般指一个Java虚拟机 ClassLoader装载的类只有一个实例，一般通过类静态属性保持该实例，这样就造成需要单例的类都需要按照单例设计模式进行编码；Spring没采用这种方式，因为该方式属于侵入式设计；代码样例如下：

```
1. package cn.javass.spring.chapter3.bean;
2. public class Singleton {
3. //1.私有化构造器
4. private Singleton() {}
5. //2.单例缓存者，惰性初始化，第一次使用时初始化
6. private static class InstanceHolder {
7. private static final Singleton INSTANCE = new Singleton();
8. }
9. //3.提供全局访问点
10. public static Singleton getInstance() {
11. return InstanceHolder.INSTANCE;
12. }
13. //4.提供一个计数器来验证一个ClassLoader一个实例
```

```
14. private int counter=0;
15. }
```

以上定义个了个单例类，首先要私有化类构造器；其次使用InstanceHolder静态内部类持有单例对象，这样可以得到惰性初始化好处；最后提供全局访问点getInstance，使得需要该单例实例的对象能获取到；我们在此还提供了一个counter计数器来验证一个ClassLoader一个实例。具体一个ClassLoader有一个单例实例测试请参考代码“cn.javass.spring.chapter3.SingletonTest”中的“testSingleton”测试方法，里边详细演示了一个ClassLoader有一个单例实例。

1） 通过注册表方式： 首先将需要单例的实例通过唯一键注册到注册表，然后通过键来获取单例，让我们直接看实现吧，注意本注册表实现了Spring接口“SingletonBeanRegistry”，该接口定义了操作共享的单例对象，Spring容器实现将实现此接口；所以共享单例对象通过“registerSingleton”方法注册，通过“getSingleton”方法获取，消除了编程方式单例，注意在实现中不考虑并发：

```
1. package cn.javass.spring.chapter3;
2. import java.util.HashMap;
3. import java.util.Map;
4. import org.springframework.beans.factory.config.SingletonBeanRegistry;
5. public class SingletonBeanRegister implements SingletonBeanRegistry {
6. //单例Bean缓存池，此处不考虑并发
7. private final Map<String, Object> BEANS = new HashMap<String, Object>();
8. public boolean containsSingleton(String beanName) {
9. return BEANS.containsKey(beanName);
10. }
11. public Object getSingleton(String beanName) {
12. return BEANS.get(beanName);
13. }
14. @Override
15. public int getSingletonCount() {
16. return BEANS.size();
17. }
18. @Override
19. public String[] getSingletonNames() {
20. return BEANS.keySet().toArray(new String[0]);
21. }
22. @Override
23. public void registerSingleton(String beanName, Object bean) {
24. if(BEANS.containsKey(beanName)) {
25. throw new RuntimeException("[" + beanName + "] 已存在");
26. }
```

```
27. BEANS.put(beanName, bean);
28. }
29. }
```

Spring是注册表单例设计模式的实现，消除了编程式单例，而且对代码是非入侵式。

接下来让我们看看在Spring中如何配置单例Bean吧，在Spring容器中如果没指定作用域默认就是“singleton”，配置方式通过scope属性配置，具体配置如下：

```
1. <bean class="cn.javass.spring.chapter3.bean.Printer" scope="singleton"/>
```

Spring管理单例对象在Spring容器中存储如图3-5所示，Spring不仅会缓存单例对象，Bean定义也是会缓存的，对于惰性初始化的对象是在首次使用时根据Bean定义创建并存放于单例缓存池。

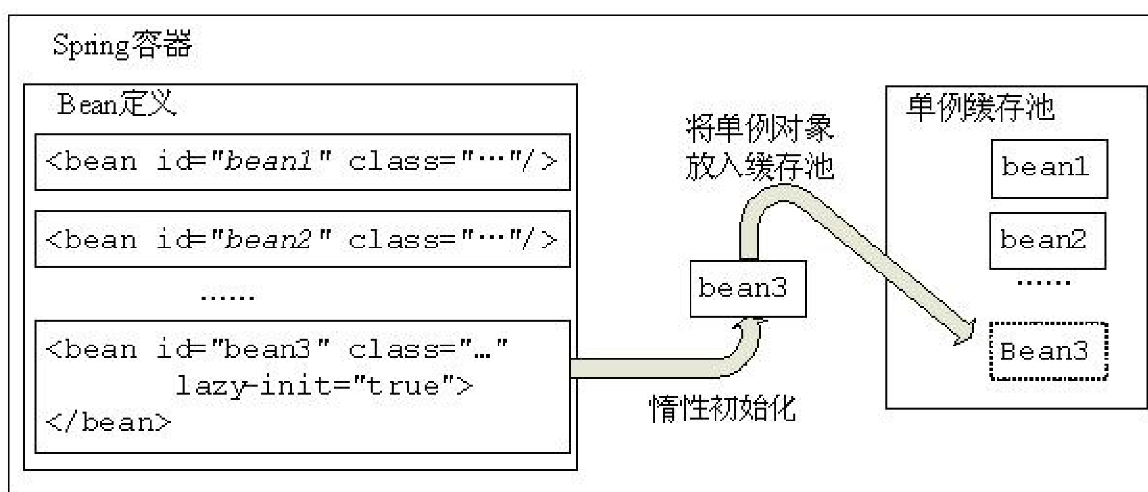


图3-5 单例处理

二、**prototype**：即原型，指每次向Spring容器请求获取Bean都返回一个全新的Bean，相对于“singleton”来说就是不缓存Bean，每次都是一个根据Bean定义创建的全新Bean。

> GoF原型设计模式，指用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

Spring中的原型和GoF中介绍的原型含义是不一样的：

GoF通过用原型实例指定创建对象的种类，而Spring容器用Bean定义指定创建对象的种类；

GoF通过拷贝这些原型创建新的对象，而Spring容器根据Bean定义创建新对象。

其相同地方都是根据某些东西创建新东西，而且GoF原型必须显示实现克隆操作，属于侵入式，而Spring容器只需配置即可，属于非侵入式。

接下来让我们看看Spring如何实现原型呢？

1）首先让我们来定义Bean“原型”：Bean定义，所有对象将根据Bean定义创建；在此我们只是简单示例一下，不会涉及依赖注入等复杂实现：BeanDefinition类定义属性“class”表示原型类，“id”表示唯一标识，“scope”表示作用域，具体如下：

```
1. package cn.javass.spring.chapter3;
2. public class BeanDefinition {
3. //单例
4. public static final int SCOPE_SINGLETON = 0;
5. //原型
6. public static final int SCOPE_PROTOTYPE = 1;
7. //唯一标识
8. private String id;
9. //class全限定名
10. private String clazz;
11. //作用域
12. private int scope = SCOPE_SINGLETON;
13. //鉴于篇幅，省略setter和getter方法；
14. }
```

2）接下来让我们看看Bean定义注册表，类似于单例注册表：

```
1. package cn.javass.spring.chapter3;
2. import java.util.HashMap;
3. import java.util.Map;
4. public class BeanDifinitionRegister {
5. //bean定义缓存，此处不考虑并发问题
6. private final Map<String, BeanDefinition> DEFINITIONS =
7. new HashMap<String, BeanDefinition>();
8. public void registerBeanDefinition(String beanName, BeanDefinition bd) {
9. //1.本实现不允许覆盖Bean定义
10. if(DEFINITIONS.containsKey(bd.getId())) {
11. throw new RuntimeException("已存在Bean定义，此实现不允许覆盖");
12. }
13. //2.将Bean定义放入Bean定义缓存池
14. DEFINITIONS.put(bd.getId(), bd);
15. }
16. public BeanDefinition getBeanDefinition(String beanName) {
17. return DEFINITIONS.get(beanName);
18. }
19. public boolean containsBeanDefinition(String beanName) {
20. return DEFINITIONS.containsKey(beanName);
21. }
```

22. }

3）接下来应该来定义BeanFactory了：

```
1. package cn.javass.spring.chapter3;
2. import org.springframework.beans.factory.config.SingletonBeanRegistry;
3. public class DefaultBeanFactory {
4. //Bean定义注册表
5. private BeanDifinitionRegister DEFINITIONS = new BeanDifinitionRegister();
6. //单例注册表
7. private final SingletonBeanRegistry SINGLETONS = new SingletonBeanRegister();
8. public Object getBean(String beanName) {
9. //1.验证Bean定义是否存在
10. if(!DEFINITIONS.containsBeanDefinition(beanName)) {
11. throw new RuntimeException("不存在[" + beanName + "]Bean定义");
12. }
13. //2.获取Bean定义
14. BeanDefinition bd = DEFINITIONS.getBeanDefinition(beanName);
15. //3.是否该Bean定义是单例作用域
16. if(bd.getScope() == BeanDefinition.SCOPE_SINGLETON) {
17. //3.1 如果单例注册表包含Bean，则直接返回该Bean
18. if(SINGLETONS.containsSingleton(beanName)) {
19. return SINGLETONS.getSingleton(beanName);
20. }
21. //3.2单例注册表不包含该Bean，
22. //则创建并注册到单例注册表，从而缓存
23. SINGLETONS.registerSingleton(beanName, createBean(bd));
24. return SINGLETONS.getSingleton(beanName);
25. }
26. //4.如果是原型Bean定义,则直接返回根据Bean定义创建的新Bean，
27. //每次都是新的，无缓存
28. if(bd.getScope() == BeanDefinition.SCOPE_PROTOTYPE) {
29. return createBean(bd);
30. }
31. //5.其他情况错误的Bean定义
32. throw new RuntimeException("错误的Bean定义");
33. }
```

```
1. public void registerBeanDefinition(BeanDefinition bd) {
2. DEFINITIONS.registerBeanDefinition(bd.getId(), bd);
```

```
3. }
4. private Object createBean(BeanDefinition bd) {
5. //根据Bean定义创建Bean
6. try {
7. Class clazz = Class.forName(bd.getClazz());
8. //通过反射使用无参数构造器创建Bean
9. return clazz.getConstructor().newInstance();
10. } catch (ClassNotFoundException e) {
11. throw new RuntimeException("没有找到Bean[" + bd.getId() + "]类");
12. } catch (Exception e) {
13. throw new RuntimeException("创建Bean[" + bd.getId() + "]失败");
14. }
15. }
```

其中方法getBean用于获取根据beanName对于的Bean定义创建的对象，有单例和原型两类Bean；registerBeanDefinition方法用于注册Bean定义，私有方法createBean用于根据Bean定义中的类型信息创建Bean。

3）测试一下吧，在此我们只测试原型作用域Bean，对于每次从Bean工厂中获取的Bean都是一个全新的对象，代码片段（BeanFatoryTest）如下：

```
1. @Test
2. public void testPrototype () throws Exception {
3. //1.创建Bean工厂
4. DefaultBeanFactory bf = new DefaultBeanFactory();
5. //2.创建原型 Bean定义
6. BeanDefinition bd = new BeanDefinition();
7. bd.setId("bean");
8. bd.setScope(BeanDefinition.SCOPE_PROTOTYPE);
9. bd.setClazz(HelloImpl2.class.getName());
10. bf.registerBeanDefinition(bd);
11. //对于原型Bean每次应该返回一个全新的Bean
12. System.out.println(bf.getBean("bean") != bf.getBean("bean"));
13. }
```

最后让我们看看如何在Spring中进行配置吧，只需指定<bean>标签属性“scope”属性为“prototype”即可：

```
1. <bean class="cn.javass.spring.chapter3.bean.Printer" scope="prototype"/>
```

Spring管理原型对象在Spring容器中存储如图3-6所示，Spring不会缓存原型对象，而是根据Bean定义每次请求返回一个全新的Bean：

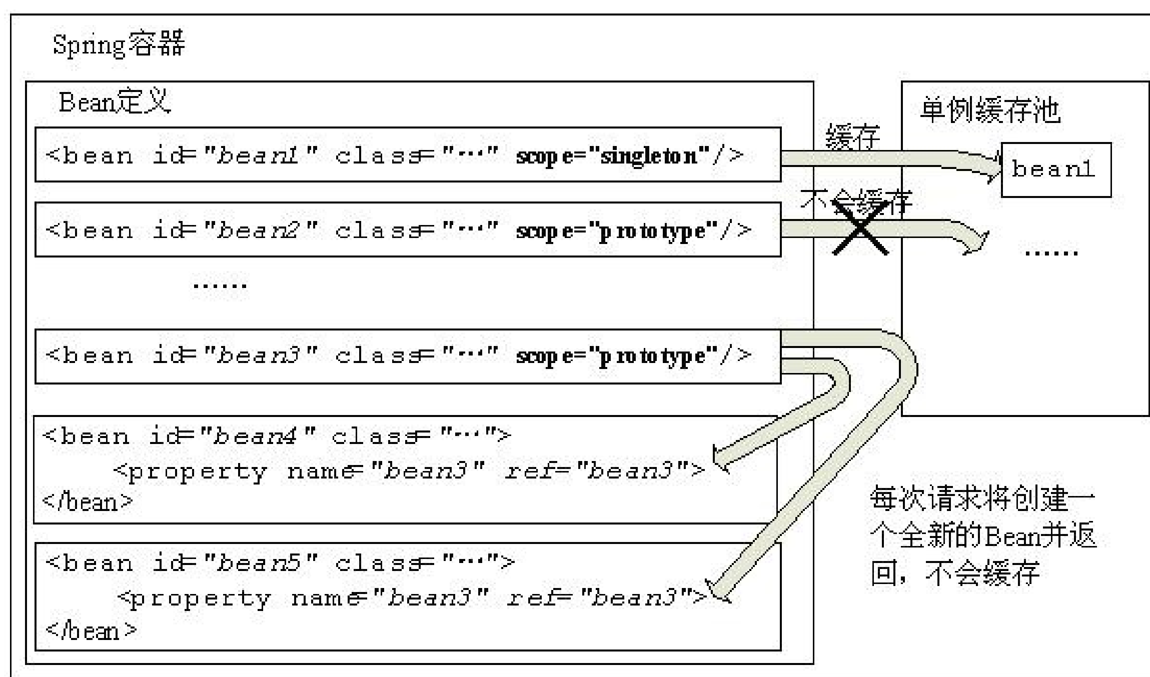


图3-6 原型处理

单例和原型作用域我们已经讲完，接下来让我们学习一些在Web应用中有哪些作用域：

3.4.2 Web应用中的作用域

在Web应用中，我们可能需要将数据存储到request、session、 session。因此Spring提供了三种Web作用域：request、session、globalSession。

一、**request**作用域：表示每个请求需要容器创建一个全新**Bean**。比如提交表单的数据必须是对每次请求新建一个Bean来保持这些表单数据，请求结束释放这些数据。

二、**session**作用域：表示每个会话需要容器创建一个全新**Bean**。比如对于每个用户一般会有一个会话，该用户的用户信息需要存储到会话中，此时可以将该Bean配置为web作用域。

三、**globalSession**：类似于session作用域，只是其用于portlet环境的web应用。如果在非portlet环境将视为session作用域。

配置方式和基本的作用域相同，只是必须要有web环境支持，并配置相应的容器监听器或拦截器从而能应用这些作用域，我们会在集成web时讲解具体使用，大家只需要知道有这些作用域就可以了。

3.4.4 自定义作用域

在日常程序开发中，几乎用不到自定义作用域，除非又必要才进行自定义作用域。

首先让我们看下Scope接口吧：

1. package org.springframework.beans.factory.config;

```
2. import org.springframework.beans.factory.ObjectFactory;
3. public interface Scope {
4. Object get(String name, ObjectFactory<?> objectFactory);
5. Object remove(String name);
6. void registerDestructionCallback(String name, Runnable callback);
7. Object resolveContextualObject(String key);
8. String getConversationId();
9. }
```

1）**Object get(String name, ObjectFactory<?> objectFactory)**：用于从作用域中获取Bean，其中参数objectFactory是当在当前作用域没找到合适Bean时使用它创建一个新的Bean；

2）**void registerDestructionCallback(String name, Runnable callback)**：用于注册销毁回调，如果想要销毁相应的对象则由Spring容器注册相应的销毁回调，而由自定义作用域选择是不是要销毁相应的对象；

3）**Object resolveContextualObject(String key)**：用于解析相应的上下文数据，比如request作用域将返回request中的属性。

4）**String getConversationId()**：作用域的会话标识，比如session作用域将是sessionId。

```
1. package cn.javass.spring.chapter3;
2. import java.util.HashMap;
3. import java.util.Map;
4. import org.springframework.beans.factory.ObjectFactory;
5. import org.springframework.beans.factory.config.Scope;
6. public class ThreadScope implements Scope {
7. private final ThreadLocal<Map<String, Object>> THREAD_SCOPE =
8. new ThreadLocal<Map<String, Object>>() {
9. protected Map<String, Object> initialValue() {
10. //用于存放线程相关Bean
11. return new HashMap<String, Object>();
12. }
13. };
```

让我们来实现个简单的thread作用域，该作用域内创建的对象将绑定到ThreadLocal内。

```
1. @Override
2. public Object get(String name, ObjectFactory<?> objectFactory) {
3. //如果当前线程已经绑定了相应Bean，直接返回
4. if(THREAD_SCOPE.get().containsKey(name)) {
5. return THREAD_SCOPE.get().get(name);
6. }
```

```
7. //使用objectFactory创建Bean并绑定到当前线程上
8. THREAD_SCOPE.get().put(name, objectFactory.getObject());
9. return THREAD_SCOPE.get().get(name);
10. }
11. @Override
12. public String getConversationId() {
13. return null;
14. }
15. @Override
16. public void registerDestructionCallback(String name, Runnable callback) {
17. //此处不实现就代表类似proytotype，容器返回给用户后就不管了
18. }
19. @Override
20. public Object remove(String name) {
21. return THREAD_SCOPE.get().remove(name);
22. }
23. @Override
24. public Object resolveContextualObject(String key) {
25. return null;
26. }
27. }
```

Scope已经实现了，让我们将其注册到Spring容器，使其发挥作用：

```
1. <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
2. <property name="scopes">
3. <map><entry>
4. <!-- 指定scope关键字 --><key><value>thread</value></key>
5. <!-- scope实现 --> <bean class="cn.javass.spring.chapter3.ThreadScope"/>
6. </entry></map>
7. </property>
8. </bean>
```

通过CustomScopeConfigurer的scopes属性注册自定义作用域实现，在此需要指定使用作用域的关键字“thread”，并指定自定义作用域实现。来让我们来定义一个“thread”作用域的Bean，配置（chapter3/threadScope.xml）如下：

```
1. <bean id="helloApi"
2. class="cn.javass.spring.chapter2.helloworld.HelloImpl"
3. scope="thread"/>
```

最后测试（cn.javass.spring.chapter3.ThreadScopeTest）一下吧，首先在一个线程中测试，在同一线程中获取的Bean应该是一样的；再让我们开启两个线程，然后应该这两个线程创建的Bean是不一样：

自定义作用域实现其实是非常简单的，其实复杂的是如果需要销毁Bean，自定义作用域如何正确的销毁Bean。

原创内容 转载请注明出处【http://sishuok.com/forum/blogPost/list/2454.html】

【第四章】 资源 之 4.1 基础知识 ——跟我学spring3

4.1.1 概述

在日常程序开发中，处理外部资源是很繁琐的事情，我们可能需要处理URL资源、File资源资源、ClassPath相关资源、服务器相关资源（JBoss AS 5.x上的VFS资源）等等很多资源。因此处理这些资源需要使用不同的接口，这就增加了我们系统的复杂性；而且处理这些资源步骤都是类似的（打开资源、读取资源、关闭资源），因此如果能抽象出一个统一的接口来对这些底层资源进行统一访问，是不是很方便，而且使我们系统更加简洁，都是对不同的底层资源使用同一个接口进行访问。

Spring 提供一个Resource接口来统一这些底层资源一致的访问，而且提供了一些便利的接口，从而能提供我们的生产力。

4.1.2 Resource接口

Spring的Resource接口代表底层外部资源，提供了对底层外部资源的一致性访问接口。

```
public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
```

```
public interface Resource extends InputStreamSource {
       boolean exists();
       boolean isReadable();
       boolean isOpen();
       URL getURL() throws IOException;
       URI getURI() throws IOException;
       File getFile() throws IOException;
       long contentLength() throws IOException;
       long lastModified() throws IOException;
       Resource createRelative(String relativePath) throws IOException;
       String getFilename();
       String getDescription();
}
```

1）InputStreamSource接口解析：

getInputStream：每次调用都将返回一个新鲜的资源对应的java.io. InputStream字节流，调用者在使用完毕后必须关闭该资源。

2）Resource接口继承InputStreamSource接口，并提供一些便利方法：

exists：返回当前Resource代表的底层资源是否存在，true表示存在。

isReadable：返回当前Resource代表的底层资源是否可读，true表示可读。

isOpen：返回当前Resource代表的底层资源是否已经打开，如果返回true，则只能被读取一次然后关闭以避免资源泄露；常见的Resource实现一般返回false。

getURL：如果当前Resource代表的底层资源能由java.util.URL代表，则返回该URL，否则抛出IOException。

getURI：如果当前Resource代表的底层资源能由java.util.URI代表，则返回该URI，否则抛出IOException。

getFile：如果当前Resource代表的底层资源能由java.io.File代表，则返回该File，否则抛出IOException。

contentLength：返回当前Resource代表的底层文件资源的长度，一般是值代表的文件资源的长度。

lastModified：返回当前Resource代表的底层资源的最后修改时间。

createRelative：用于创建相对于当前Resource代表的底层资源的资源，比如当前Resource代表文件资源"d:/test/"则createRelative（"test.txt"）将返回表文件资源"d:/test/test.txt"Resource资源。

getFilename：返回当前Resource代表的底层文件资源的文件路径，比如File资源"file://d:/test.txt"将返回"d:/test.txt"，而URL资源http://www.javass.cn将返回""，因为只返回文件路径。

getDescription：返回当前Resource代表的底层资源的描述符，通常就是资源的全路径（实际文件名或实际URL地址）。

Resource接口提供了足够的抽象，足够满足我们日常使用。而且提供了很多内置Resource实现：ByteArrayResource、InputStreamResource 、FileSystemResource 、UrlResource 、ClassPathResource、ServletContextResource、VfsResource等。

原创内容 转自请注明【http://sishuok.com/forum/blogPost/list/0/2455.html】

【第四章】 资源 之 4.2 内置Resource实现 ——跟我学spring3

4.2 内置Resource实现

4.2.1 ByteArrayResource

ByteArrayResource代表byte[]数组资源，对于“getInputStream”操作将返回一个ByteArrayInputStream。

首先让我们看下使用ByteArrayResource如何处理byte数组资源：

```
package cn.javass.spring.chapter4;
import java.io.IOException;
import java.io.InputStream;
import org.junit.Test;
import org.springframework.core.io.ByteArrayResource;
import org.springframework.core.io.Resource;
public class ResourceTest {
@Test
public void testByteArrayResource() {
Resource resource = new ByteArrayResource("Hello World!".getBytes());
        if(resource.exists()) {
            dumpStream(resource);
        }
}
}
```

是不是很简单，让我们看下“dumpStream”实现：

```
private void dumpStream(Resource resource) {
        InputStream is = null;
        try {
            //1.获取文件资源
            is = resource.getInputStream();
            //2.读取资源
            byte[] descBytes = new byte[is.available()];
            is.read(descBytes);
            System.out.println(new String(descBytes));
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            try {
                //3.关闭资源
                is.close();
            } catch (IOException e) {
            }
        }
    }
```

让我们来仔细看一下代码，dumpStream方法很抽象定义了访问流的三部曲：打开资源、读取资源、关闭资源，所以dunpStrean可以再进行抽象从而能在自己项目中使用；byteArrayResourceTest测试方法，也定义了基本步骤：定义资源、验证资源存在、访问资源。

ByteArrayResource可多次读取数组资源，即isOpen ()永远返回false。

1.2.2 InputStreamResource

InputStreamResource代表java.io.InputStream字节流，对于“getInputStream ”操作将直接返回该字节流，因此只能读取一次该字节流，即“isOpen”永远返回true。

让我们看下测试代码吧：

```
@Test
public void testInputStreamResource() {
   ByteArrayInputStream bis = new ByteArrayInputStream("Hello World!".getBytes());
   Resource resource = new InputStreamResource(bis);
    if(resource.exists()) {
       dumpStream(resource);
    }
    Assert.assertEquals(true, resource.isOpen());
}
```

测试代码几乎和ByteArrayResource测试完全一样，注意“isOpen”此处用于返回true。

4.2.3 FileSystemResource

FileSystemResource代表java.io.File资源，对于“getInputStream ”操作将返回底层文件的字节流，“isOpen”将永远返回false，从而表示可多次读取底层文件的字节流。

让我们看下测试代码吧：

```
@Test
public void testFileResource() {
File file = new File("d:/test.txt");
    Resource resource = new FileSystemResource(file);
    if(resource.exists()) {
        dumpStream(resource);
    }
    Assert.assertEquals(false, resource.isOpen());
}
```

注意由于“isOpen”将永远返回false，所以可以多次调用dumpStream(resource)。

4.2.4 ClassPathResource

ClassPathResource代表classpath路径的资源，将使用ClassLoader进行加载资源。classpath 资源存在于类路径中的文件系统中或jar包里，且“isOpen”永远返回false，表示可多次读取资源。

ClassPathResource加载资源替代了Class类和ClassLoader类的“(name)”和“(name)”两个加载类路径资源方法，提供一致的访问方式。

ClassPathResource提供了三个构造器：

public ClassPathResource(String path)：使用默认的ClassLoader加载“path”类路径资源；

public ClassPathResource(String path, ClassLoader classLoader)：使用指定的ClassLoader加载“path”类路径资源；

比如当前类路径是“cn.javass.spring.chapter4.ResourceTest”，而需要加载的资源路径是“cn/javass/spring/chapter4/test1.properties”，则将加载的资源在“cn/javass/spring/chapter4/test1.properties”；

public ClassPathResource(String path, Class<?> clazz)：使用指定的类加载“path”类路径资源，将加载相对于当前类的路径的资源；

比如当前类路径是“cn.javass.spring.chapter4.ResourceTest”，而需要加载的资源路径是“cn/javass/spring/chapter4/test1.properties”，则将加载的资源在“cn/javass/spring/chapter4/cn/javass/spring/chapter4/test1.properties”；

而如果需要 加载的资源路径为“test1.properties”，将加载的资源为“cn/javass/spring/chapter4/test1.properties”。

让我们直接看测试代码吧：

1）使用默认的加载器加载资源，将加载当前ClassLoader类路径上相对于根路径的资源：

```
@Test
public void testClasspathResourceByDefaultClassLoader() throws IOException {
   Resource resource = new ClassPathResource("cn/javass/spring/chapter4/test1.properties"
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

2）使用指定的ClassLoader进行加载资源，将加载指定的ClassLoader类路径上相对于根路径的资源：

```
@Test
public void testClasspathResourceByClassLoader() throws IOException {
    ClassLoader cl = this.getClass().getClassLoader();
    Resource resource = new ClassPathResource("cn/javass/spring/chapter4/test1.properties
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

3）使用指定的类进行加载资源，将尝试加载相对于当前类的路径的资源：

```
@Test
public void testClasspathResourceByClass() throws IOException {
   Class clazz = this.getClass();
    Resource resource1 = new ClassPathResource("cn/javass/spring/chapter4/test1.propertie
    if(resource1.exists()) {
        dumpStream(resource1);
    }
    System.out.println("path:" + resource1.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource1.isOpen());

    Resource resource2 = new ClassPathResource("test1.properties" , this.getClass());
    if(resource2.exists()) {
        dumpStream(resource2);
   }
    System.out.println("path:" + resource2.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource2.isOpen());
}
```

“resource1”将加载cn/javass/spring/chapter4/cn/javass/spring/chapter4/test1.properties资源；“resource2”将加载“cn/javass/spring/chapter4/test1.properties”；

4）加载jar包里的资源，首先在当前类路径下找不到，最后才到Jar包里找，而且在第一个Jar包里找到的将被返回：

```
@Test
public void classpathResourceTestFromJar() throws IOException {
Resource resource = new ClassPathResource("overview.html");
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getURL().getPath());
    Assert.assertEquals(false, resource.isOpen());
}
```

如果当前类路径包含“overview.html”，在项目的“resources”目录下，将加载该资源，否则将加载Jar包里的“overview.html”，而且不能使用“resource.getFile()”，应该使用“resource.getURL()”，因为资源不存在于文件系统而是存在于jar包里，URL类似于“file:/C:/.../*.jar!/overview.html”。

类路径一般都是相对路径，即相对于类路径或相对于当前类的路径，因此如果使用“/test1.properties”带前缀“/”的路径，将自动删除“/”得到“test1.properties”。

4.2.5 UrlResource

UrlResource代表URL资源，用于简化URL资源访问。“isOpen”永远返回false，表示可多次读取资源。

UrlResource一般支持如下资源访问：

http：通过标准的http协议访问web资源，如new UrlResource(“http://地址”)；

ftp：通过ftp协议访问资源，如new UrlResource(“ftp://地址”)；

file：通过file协议访问本地文件系统资源，如new UrlResource(“file:d:/test.txt”)；

具体使用方法在此就不演示了，可以参考cn.javass.spring.chapter4.ResourceTest中urlResourceTest测试方法。

4.2.6 ServletContextResource

ServletContextResource代表web应用资源，用于简化servlet容器的ServletContext接口的getResource操作和getResourceAsStream操作；在此就不具体演示了。

4.2.7 VfsResource

VfsResource代表Jboss 虚拟文件系统资源。

Jboss VFS(Virtual File System)框架是一个文件系统资源访问的抽象层，它能一致的访问物理文件系统、jar资源、zip资源、war资源等，VFS能把这些资源一致的映射到一个目录上，访问它们就像访问物理文件资源一样，而其实这些资源不存在于物理文件系统。

在示例之前需要准备一些jar包，在此我们使用的是Jboss VFS3版本，可以下载最新的Jboss AS 6x，拷贝lib目录下的“jboss-logging.jar”和“jboss-vfs.jar”两个jar包拷贝到我们项目的lib目录中并添加到“Java Build Path”中的“Libaries”中。

让我们看下示例（cn.javass.spring.chapter4.ResourceTest）：

```
@Test
public void testVfsResourceForRealFileSystem() throws IOException {
//1.创建一个虚拟的文件目录
VirtualFile home = VFS.getChild("/home");
//2.将虚拟目录映射到物理的目录
VFS.mount(home, new RealFileSystem(new File("d:")));
//3.通过虚拟目录获取文件资源
VirtualFile testFile = home.getChild("test.txt");
//4.通过一致的接口访问
Resource resource = new VfsResource(testFile);
if(resource.exists()) {
        dumpStream(resource);
}
System.out.println("path:" + resource.getFile().getAbsolutePath());
Assert.assertEquals(false, resource.isOpen());
}
@Test
public void testVfsResourceForJar() throws IOException {
//1.首先获取jar包路径
    File realFile = new File("lib/org.springframework.beans-3.0.5.RELEASE.jar");
    //2.创建一个虚拟的文件目录
    VirtualFile home = VFS.getChild("/home2");
    //3.将虚拟目录映射到物理的目录
VFS.mountZipExpanded(realFile, home,
TempFileProvider.create("tmp", Executors.newScheduledThreadPool(1)));
//4.通过虚拟目录获取文件资源
    VirtualFile testFile = home.getChild("META-INF/spring.handlers");
    Resource resource = new VfsResource(testFile);
    if(resource.exists()) {
            dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

通过VFS，对于jar里的资源和物理文件系统访问都具有一致性，此处只是简单示例，如果需要请到Jboss官网深入学习。

原创内容 转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2456.html】

【第四章】 资源 之 4.3 访问Resource ——跟我学spring3

4.3.1 ResourceLoader接口

ResourceLoader接口用于返回Resource对象；其实现可以看作是一个生产Resource的工厂类。

```
public interface ResourceLoader {
       Resource getResource(String location);
       ClassLoader getClassLoader();
}
```

getResource接口用于根据提供的location参数返回相应的Resource对象；而getClassLoader则返回加载这些Resource的ClassLoader。

Spring提供了一个适用于所有环境的DefaultResourceLoader实现，可以返回ClassPathResource、UrlResource；还提供一个用于web环境的ServletContextResourceLoader，它继承了DefaultResourceLoader的所有功能，又额外提供了获取ServletContextResource的支持。

ResourceLoader在进行加载资源时需要使用前缀来指定需要加载："classpath:path"表示返回ClasspathResource，"http://path"和"file:path"表示返回UrlResource资源，如果不加前缀则需要根据当前上下文来决定，DefaultResourceLoader默认实现可以加载classpath资源，如代码所示（cn.javass.spring.chapter4.ResourceLoaderTest）：

```
@Test
public void testResourceLoad() {
    ResourceLoader loader = new DefaultResourceLoader();
    Resource resource = loader.getResource("classpath:cn/javass/spring/chapter4/test1.txt
    //验证返回的是ClassPathResource
    Assert.assertEquals(ClassPathResource.class, resource.getClass());
    Resource resource2 = loader.getResource("file:cn/javass/spring/chapter4/test1.txt");
    //验证返回的是ClassPathResource
    Assert.assertEquals(UrlResource.class, resource2.getClass());
    Resource resource3 = loader.getResource("cn/javass/spring/chapter4/test1.txt");
    //验证返默认可以加载ClasspathResource
    Assert.assertTrue(resource3 instanceof ClassPathResource);
}
```

对于目前所有ApplicationContext都实现了ResourceLoader，因此可以使用其来加载资源。

ClassPathXmlApplicationContext：不指定前缀将返回默认的ClassPathResource资源，否则将根据前缀来加载资源；

FileSystemXmlApplicationContext：不指定前缀将返回FileSystemResource，否则将根据前缀来加载资源；

WebApplicationContext：不指定前缀将返回ServletContextResource，否则将根据前缀来加载资源；

其他：不指定前缀根据当前上下文返回Resource实现，否则将根据前缀来加载资源。

4.3.2 ResourceLoaderAware接口

ResourceLoaderAware是一个标记接口，用于通过ApplicationContext上下文注入ResourceLoader。

```
public interface ResourceLoaderAware {
   void setResourceLoader(ResourceLoader resourceLoader);
}
```

让我们看下测试代码吧：

1） 首先准备测试Bean，我们的测试Bean还简单只需实现ResourceLoaderAware接口，然后通过回调将ResourceLoader保存下来就可以了：

```
package cn.javass.spring.chapter4.bean;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.core.io.ResourceLoader;
public class ResourceBean implements ResourceLoaderAware {
    private ResourceLoader resourceLoader;
    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }
    public ResourceLoader getResourceLoader() {
        return resourceLoader;
    }
}
```

2） 配置Bean定义（chapter4/resourceLoaderAware.xml）：

```
&lt;bean class="cn.javass.spring.chapter4.bean.ResourceBean"/&gt;
```

3）测试(cn.javass.spring.chapter4.ResoureLoaderAwareTest)：

```
@Test
public void test() {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter4/resourceLoaderA
    ResourceBean resourceBean = ctx.getBean(ResourceBean.class);
    ResourceLoader loader = resourceBean.getResourceLoader();
    Assert.assertTrue(loader instanceof ApplicationContext);
}
```

注意此处“loader instanceof ApplicationContext”，说明了ApplicationContext就是个ResoureLoader。

由于上述实现回调接口注入ResourceLoader的方式属于侵入式，所以不推荐上述方法，可以采用更好的自动注入方式，如“byType”和“constructor”，此处就不演示了。

4.3.3 注入Resource

通过回调或注入方式注入“ResourceLoader”，然后再通过“ResourceLoader”再来加载需要的资源对于只需要加载某个固定的资源是不是很麻烦，有没有更好的方法类似于前边实例中注入“java.io.File”类似方式呢？

Spring提供了一个PropertyEditor “ResourceEditor”用于在注入的字符串和Resource之间进行转换。因此可以使用注入方式注入Resource。

ResourceEditor完全使用ApplicationContext根据注入的路径字符串获取相应的Resource，说白了还是自己做还是容器帮你做的问题。

接下让我们看下示例：

1）准备Bean：

```
package cn.javass.spring.chapter4.bean;
import org.springframework.core.io.Resource;
public class ResourceBean3 {
    private Resource resource;
    public Resource getResource() {
        return resource;
    }
    public void setResource(Resource resource) {
        this.resource = resource;
    }
}
```

2）准备配置文件（chapter4/ resourceInject.xml）：

```
&lt;bean id="resourceBean1" class="cn.javass.spring.chapter4.bean.ResourceBean3"&gt;
   &lt;property name="resource" value="cn/javass/spring/chapter4/test1.properties"/&gt;
&lt;/bean&gt;
&lt;bean id="resourceBean2" class="cn.javass.spring.chapter4.bean.ResourceBean3"&gt;
&lt;property name="resource"
value="classpath:cn/javass/spring/chapter4/test1.properties"/&gt;
&lt;/bean&gt;
```

注意此处“resourceBean1”注入的路径没有前缀表示根据使用的ApplicationContext实现进行选择Resource实现。

3）让我们来看下测试代码（cn.javass.spring.chapter4.ResourceInjectTest）吧：

```
@Test
public void test() {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter4/resourceInject.
    ResourceBean3 resourceBean1 = ctx.getBean("resourceBean1", ResourceBean3.class);
    ResourceBean3 resourceBean2 = ctx.getBean("resourceBean2", ResourceBean3.class);
    Assert.assertTrue(resourceBean1.getResource() instanceof ClassPathResource);
    Assert.assertTrue(resourceBean2.getResource() instanceof ClassPathResource);
}
```

接下来一节让我们深入ApplicationContext对各种Resource的支持，及如何使用更便利的资源加载方式。

原创内容 转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2457.html】

【第四章】 资源 之 4.4 Resource通配符路径 ——跟我学spring3

4.4.1 使用路径通配符加载Resource

前面介绍的资源路径都是非常简单的一个路径匹配一个资源，Spring还提供了一种更强大的Ant模式通配符匹配，从能一个路径匹配一批资源。

Ant路径通配符支持“？”、“”、“*”，注意通配符匹配不包括目录分隔符“/”：

“?”：匹配一个字符，如“config?.xml”将匹配“config1.xml”；

“*”：匹配零个或多个字符串，如“cn/*/config.xml”将匹配*“cn/javass/config.xml”*，但不匹配匹配*“cn/config.xml”*；而*“cn/config*-.xml”将匹配“cn/config-dao.xml”；

“”：匹配路径中的零个或多个目录，如**“cn/**/config.xml”将匹配“cn /config.xml”，也匹配“cn/javass/spring/config.xml”；而“cn/javass/config-**.xml”**将匹配**“cn/javass/config-dao.xml”**，即把**“”**当做两个“*”处理。

Spring提供AntPathMatcher来进行Ant风格的路径匹配。具体测试请参考cn.javass.spring.chapter4. AntPathMatcherTest。

Spring在加载类路径资源时除了提供前缀“classpath:”的来支持加载一个Resource，还提供一个前缀“classpath*:”来支持加载所有匹配的类路径Resource。

Spring提供ResourcePatternResolver接口来加载多个Resource，该接口继承了ResourceLoader并添加了“Resource[] getResources(String locationPattern)”用来加载多个Resource：

```
1.  public interface ResourcePatternResolver extends ResourceLoader {
2.  String CLASSPATH_ALL_URL_PREFIX = "classpath*:";
3.  Resource[] getResources(String locationPattern) throws IOException;
4.  }
```

Spring提供了一个ResourcePatternResolver实现PathMatchingResourcePatternResolver，它是基于模式匹配的，默认使用AntPathMatcher进行路径匹配，它除了支持ResourceLoader支持的前缀外，还额外支持“classpath*:”用于加载所有匹配的类路径*Resource*，*ResourceLoader*不支持前缀*“classpath*:”：

首先做下准备工作，在项目的“resources”创建“META-INF”目录，然后在其下创建一个“INDEX.LIST”文件。同时在“org.springframework.beans-3.0.5.RELEASE.jar”和“org.springframework.context-3.0.5.RELEASE.jar”两个jar包里也存在

相同目录和文件。然后创建一个“LICENSE”文件，该文件存在于“com.springsource.cn.sf.cglib-2.2.0.jar”里。

一、**“classpath”**：用于加载类路径（包括jar包）中的一个且仅一个资源；对于多个匹配的也只返回一个，所以如果需要多个匹配的请考虑“classpath*:”前缀；

```
1. @Test
2. public void testClasspathPrefix() throws IOException {
3. ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
4. //只加载一个绝对匹配Resource，且通过ResourceLoader.getResource进行加载
5. Resource[] resources=resolver.getResources("classpath:META-INF/INDEX.LIST");
6. Assert.assertEquals(1, resources.length);
7. //只加载一个匹配的Resource，且通过ResourceLoader.getResource进行加载
8. resources = resolver.getResources("classpath:META-INF/*.LIST");
9. Assert.assertTrue(resources.length == 1);
10. }
```

二、**“classpath*”**：用于加载类路径（包括jar包）中的所有匹配的资源。带通配符的classpath使用“ClassLoader”的“<> (name)”方法来查找通配符之前的资源，然后通过模式匹配来获取匹配的资源。如“classpath:META-INF/*.LIST”将首先加载通配符之前的目录“META-INF”，然后再遍历路径进行子路径匹配从而获取匹配的资源。

```
1. @Test
2. public void testClasspathAsteriskPrefix () throws IOException {
3. ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
4. //将加载多个绝对匹配的所有Resource
5. //将首先通过ClassLoader.getResources("META-INF")加载非模式路径部分
6. //然后进行遍历模式匹配
7. Resource[] resources=resolver.getResources("classpath*:META-INF/INDEX.LIST");
8. Assert.assertTrue(resources.length > 1);
9. //将加载多个模式匹配的Resource
10. resources = resolver.getResources("classpath:META-INF/.LIST");
11. Assert.assertTrue(resources.length > 1);
12. }
```

注意“resources.length >1”说明返回多个Resource。不管模式匹配还是非模式匹配只要匹配的都将返回。

在“com.springsource.cn.sf.cglib-2.2.0.jar”里包含“asm-license.txt”文件，对于使用“classpath*:asm-*.txt*”进行通配符方式加载资源将什么也加载不了“asm-license.txt”文件，注意一定是模式路径匹配才会遇到这种问题。这是由于“ClassLoader”的“(name)”方法的限制，对于name为“”的情况将只返回文件系统的类路径，不会包换jar包根路径。

```
1. @Test
```

```
2. public void testClasspathAsteriskPrefixLimit() throws IOException {
3. ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver(); //将首先通过ClassLoader.getResources("")加载目录，
4. //将只返回文件系统的类路径不返回jar的跟路径
5. //然后进行遍历模式匹配
6. Resource[] resources = resolver.getResources("classpath:asm-.txt");
7. Assert.assertTrue(resources.length == 0);
8. //将通过ClassLoader.getResources("asm-license.txt")加载
9. //asm-license.txt存在于com.springsource.net.sf.cglib-2.2.0.jar
10. resources = resolver.getResources("classpath*:asm-license.txt");
11. Assert.assertTrue(resources.length > 0);
12. //将只加载文件系统类路径匹配的Resource
13. resources = resolver.getResources("classpath:LICENS");
14. Assert.assertTrue(resources.length == 1);
15. }
```

对于“resolver.getResources("classpath:*asm*-.txt");”，由于在项目“resources”目录下没有所以应该返回0个资源；“resolver.getResources("classpath:*asm-license.txt");”*将返回*jar*包里的*Resource*；*“resolver.getResources("classpath*:LICENS*");”，因为将只返回文件系统类路径资源，所以返回1个资源。

因此加载通配符路径时（即路径中包含通配符），必须包含一个根目录才能保证加载的资源是所有的，而不是部分。

三、**“file”**：加载一个或多个文件系统中的Resource。如“file:D:/*.txt”将返回D盘下的所有txt文件；

四、无前缀：通过ResourceLoader实现加载一个资源。

AppliacationContext提供的getResources方法将获取资源委托给ResourcePatternResolver实现，默认使用PathMatchingResourcePatternResolver。所有在此就无需介绍其使用方法了。

4.4.2 注入Resource数组

Spring还支持注入Resource数组，直接看配置如下：

```
1. <bean id="resourceBean1" class="cn.javass.spring.chapter4.bean.ResourceBean4">
2. <property name="resources">
3. <array>
4. <value>cn/javass/spring/chapter4/test1.properties</value>
5. <value>log4j.xml</value>
6. </array>
7. </property>
```

```
8. </bean>
9. <bean id="resourceBean2" class="cn.javass.spring.chapter4.bean.ResourceBean4">
10. <property name="resources" value="classpath*:META-INF/INDEX.LIST"/>
11. </bean>
12. <bean id="resourceBean3" class="cn.javass.spring.chapter4.bean.ResourceBean4">
13. <property name="resources">
14. <array>
15. <value>cn/javass/spring/chapter4/test1.properties</value>
16. <value>classpath*:META-INF/INDEX.LIST</value>
17. </array>
18. </property>
19. </bean>
```

“resourceBean1”就不用多介绍了，传统实现方式；对于“resourceBean2”则使用前缀“classpath*”，看到这大家应该懂的，加载匹配多个资源；“resourceBean3”是混合使用的；测试代码在“cn.javass.spring.chapter4.ResourceInjectTest.testResourceArrayInject”。

Spring通过ResourceArrayPropertyEditor来进行类型转换的，而它又默认使用“PathMatchingResourcePatternResolver”来进行把路径解析为Resource对象。所有大家只要会使用“PathMatchingResourcePatternResolver”，其它一些实现都是委托给它的，比如AppliacationContext的“getResources”方法等。

4.4.3 AppliacationContext实现对各种Resource的支持

一、**ClassPathXmlApplicationContext**：默认将通过classpath进行加载返回ClassPathResource，提供两类构造器方法：

```
1. public class ClassPathXmlApplicationContext {
2. //1）通过ResourcePatternResolver实现根据configLocation获取资源
3. public ClassPathXmlApplicationContext(String configLocation);
4. public ClassPathXmlApplicationContext(String... configLocations)；
5. public ClassPathXmlApplicationContext(String[] configLocations, ……);

6. //2）通过直接根据path直接返回ClasspathResource

7. public ClassPathXmlApplicationContext(String path, Class clazz);
8. public ClassPathXmlApplicationContext(String[] paths, Class clazz);
9. public ClassPathXmlApplicationContext(String[] paths, Class clazz, ……);
10. }
```

第一类构造器是根据提供的配置文件路径使用“ResourcePatternResolver”的“getResources()”接口通过匹配获取资源；即如“classpath:config.xml”

第二类构造器则是根据提供的路径和clazz来构造ClassResource资源。即采用“public ClassPathResource(String path, Class<?> clazz)”构造器获取资源。

二、**FileSystemXmlApplicationContext**：将加载相对于当前工作目录的“configLocation”位置的资源，注意在linux系统上不管“configLocation”是否带“/”，都作为相对路径；而在window系统上如“D:/resourceInject.xml”是绝对路径。因此在除非很必要的情况下，不建议使用该ApplicationContext。

```
1. public class FileSystemXmlApplicationContext{
2. public FileSystemXmlApplicationContext(String configLocation);
3. public FileSystemXmlApplicationContext(String... configLocations,……);
4. }
```

```
1. //linux系统，以下全是相对于当前vm路径进行加载
2. new FileSystemXmlApplicationContext("chapter4/config.xml");
3. new FileSystemXmlApplicationContext("/chapter4/confg.xml");
```

```
1. //windows系统，第一个将相对于当前vm路径进行加载；
2. //第二个则是绝对路径方式加载
3. new FileSystemXmlApplicationContext("chapter4/config.xml");
4. new FileSystemXmlApplicationContext("d:/chapter4/confg.xml");
```

此处还需要注意：在linux系统上，构造器使用的是相对路径，而ctx.getResource()方法如果以“/”开头则表示获取绝对路径资源，而不带前导“/”将返回相对路径资源。如下：

```
1. //linux系统，第一个将相对于当前vm路径进行加载；
2. //第二个则是绝对路径方式加载
3. ctx.getResource ("chapter4/config.xml");
4. ctx.getResource ("/root/confg.xml");
5. //windows系统，第一个将相对于当前vm路径进行加载；
6. //第二个则是绝对路径方式加载
7. ctx.getResource ("chapter4/config.xml");
8. ctx.getResource ("d:/chapter4/confg.xml");
```

因此如果需要加载绝对路径资源最好选择前缀“file”方式，将全部根据绝对路径加载。如在linux系统“ctx.getResource ("file:/root/confg.xml");”

原创内容，转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2458.html】

【第五章】Spring表达式语言 之 5.1 概述 5.2 SpEL基础 ——跟我学spring3

5.1 概述

5.1.1 概述

Spring表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于Struts2x中使用的OGNL表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与Spring功能完美整合，如能用来配置Bean定义。

表达式语言给静态Java语言增加了动态功能。

SpEL是单独模块，只依赖于core模块，不依赖于其他模块，可以单独使用。

5.1.2 能干什么

表达式语言一般是用最简单的形式完成最主要的工作，减少我们的工作量。

SpEL支持如下表达式：

一、基本表达式：字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及Elivis表达式、正则表达式、括号优先级表达式；

二、类相关表达式：类类型表达式、类实例化、instanceof表达式、变量定义及引用、赋值表达式、自定义函数、对象属性存取及安全导航表达式、对象方法调用、Bean引用；

三、集合相关表达式：内联List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

四、其他表达式：模板表达式。

注：**SpEL**表达式中的关键字是不区分大小写的。

5.2 SpEL基础

5.2.1 HelloWorld

首先准备支持SpEL的Jar包：“org.springframework.expression-3.0.5.RELEASE.jar”将其添加到类路径中。

SpEL在求表达式值时一般分为四步，其中第三步可选：首先构造一个解析器，其次解析器解析字符串表达式，在此构造上下文，最后根据上下文得到表达式运算后的值。

让我们看下代码片段吧：

```
package cn.javass.spring.chapter5;
import junit.framework.Assert;
import org.junit.Test;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
public class SpELTest {
    @Test
    public void helloWorld() {
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression =
parser.parseExpression("('Hello' + ' World').concat(#end)");
        EvaluationContext context = new StandardEvaluationContext();
        context.setVariable("end", "!");
        Assert.assertEquals("Hello World!", expression.getValue(context));
    }
}
```

接下来让我们分析下代码：

1）创建解析器：SpEL使用ExpressionParser接口表示解析器，提供SpelExpressionParser默认实现；

2）解析表达式：使用ExpressionParser的parseExpression来解析相应的表达式为Expression对象。

3）构造上下文：准备比如变量定义等等表达式需要的上下文数据。

4）求值：通过Expression接口的getValue方法根据上下文获得表达式值。

是不是很简单，接下来让我们看下其具体实现及原理吧。

5.2.3 SpEL原理及接口

SpEL提供简单的接口从而简化用户使用，在介绍原理前让我们学习下几个概念：

一、表达式：表达式是表达式语言的核心，所以表达式语言都是围绕表达式进行的，从我们角度来看是“干什么”；

二、解析器：用于将字符串表达式解析为表达式对象，从我们角度来看是“谁来干”；

三、上下文：表达式对象执行的环境，该环境可能定义变量、定义自定义函数、提供类型转换等等，从我们角度看是“在哪干”；

四、根对象及活动上下文对象：根对象是默认的活动上下文对象，活动上下文对象表示了当前表达式操作的对象，从我们角度看是“对谁干”。

理解了这些概念后，让我们看下SpEL如何工作的呢，如图5-1所示：

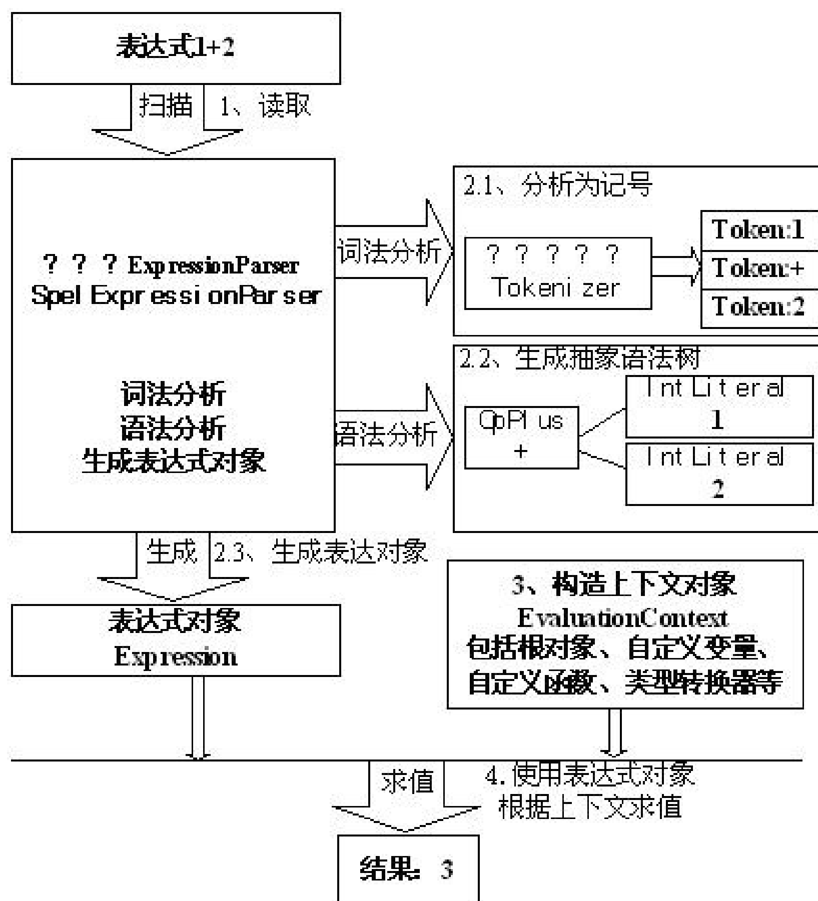


图5-1 工作原理

1）首先定义表达式："1+2"；

2）定义解析器ExpressionParser实现，SpEL提供默认实现SpelExpressionParser；

2.1）SpelExpressionParser解析器内部使用Tokenizer类进行词法分析，即把字符串流分析为记号流，记号在SpEL使用Token类来表示；

2.2）有了记号流后，解析器便可根据记号流生成内部抽象语法树；在SpEL中语法树节点由SpelNode接口实现代表：如OpPlus表示加操作节点、IntLiteral表示int型字面量节点；使用SpelNodel实现组成了抽象语法树；

2.3）对外提供Expression接口来简化表示抽象语法树，从而隐藏内部实现细节，并提供getValue简单方法用于获取表达式值；SpEL提供默认实现为SpelExpression；

3）定义表达式上下文对象（可选），SpEL使用EvaluationContext接口表示上下文对象，用于设置根对象、自定义变量、自定义函数、类型转换器等，SpEL提供默认实现StandardEvaluationContext；

4）使用表达式对象根据上下文对象（可选）求值（调用表达式对象的getValue方法）获得结果。

接下来让我们看下SpEL的主要接口吧：

1）**ExpressionParser**接口：表示解析器，默认实现是org.springframework.expression.spel.standard包中的SpelExpressionParser类，使用parseExpression方法将字符串表达式转换为Expression对象，对于ParserContext接口用于定义字符串表达式是不是模板，及模板开始与结束字符：

```
public interface ExpressionParser {
       Expression parseExpression(String expressionString);
       Expression parseExpression(String expressionString, ParserContext context);
}
```

来看下示例：

```
@Test
public void testParserContext() {
    ExpressionParser parser = new SpelExpressionParser();
    ParserContext parserContext = new ParserContext() {
        @Override
         public boolean isTemplate() {
            return true;
        }
        @Override
        public String getExpressionPrefix() {
            return "#{";
        }
        @Override
        public String getExpressionSuffix() {
            return "}";
        }
    };
    String template = "#{'Hello '}#{'World!'}";
    Expression expression = parser.parseExpression(template, parserContext);
    Assert.assertEquals("Hello World!", expression.getValue());
}
```

在此我们演示的是使用ParserContext的情况，此处定义了ParserContext实现：定义表达式是模块，表达式前缀为“#{”，后缀为“}”；使用parseExpression解析时传入的模板必须以“#{”开头，以“}”结尾，如"#{'Hello '}#{'World!'}"。

默认传入的字符串表达式不是模板形式，如之前演示的Hello World。

2）**EvaluationContext**接口：表示上下文环境，默认实现是org.springframework.expression.spel.support包中的StandardEvaluationContext类，使用setRootObject方法来设置根对象，使用setVariable方法来注册自定义变量，使用registerFunction来注册自定义函数等等。

3）**Expression**接口：表示表达式对象，默认实现是org.springframework.expression.spel.standard包中的SpelExpression，提供getValue方法用于获取表达式值，提供setValue方法用于设置对象值。

了解了SpEL原理及接口，接下来的事情就是SpEL语法了。

【第五章】Spring表达式语言 之 5.3 SpEL语法 ——跟我学spring3

5.3 SpEL语法

5.3.1 基本表达式

一、字面量表达式：SpEL支持的字面量包括：字符串、数字类型（int、long、float、double）、布尔类型、null类型。

类型	示例
字符串	String str1 = parser.parseExpression("'Hello World!'").getValue(String.class);String str2 = parser.parseExpression("\"Hello World!\"").getValue(String.class);
数字类型	int int1 = parser.parseExpression("1").getValue(Integer.class);long long1 = parser.parseExpression("-1L").getValue(long.class);float float1 = parser.parseExpression("1.1").getValue(Float.class);double double1 = parser.parseExpression("1.1E+2").getValue(double.class);int hex1 = parser.parseExpression("0xa").getValue(Integer.class);long hex2 = parser.parseExpression("0xaL").getValue(long.class);
布尔类型	boolean true1 = parser.parseExpression("true").getValue(boolean.class);boolean false1 = parser.parseExpression("false").getValue(boolean.class);
null类型	Object null1 = parser.parseExpression("null").getValue(Object.class);

二、算数运算表达式：SpEL支持加(+)、减(-)、乘(*)、除(/)、求余（%）、幂（^）运算。

类型	示例
加减乘除	int result1 = parser.parseExpression("1+2-3*4/2").getValue(Integer.class);//-3
求余	int result2 = parser.parseExpression("4%3").getValue(Integer.class);//1
幂运算	int result3 = parser.parseExpression("2^3").getValue(Integer.class);//8

SpEL还提供求余（MOD）和除（DIV）而外两个运算符，与“%”和“/”等价，不区分大小写。

三、关系表达式：等于（==）、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)，区间（between）运算，如“parser.parseExpression("1>2").getValue(boolean.class);”将返回false；而“parser.parseExpression("1 between {1, 2}").getValue(boolean.class);”将返回true。

between运算符右边操作数必须是列表类型，且只能包含2个元素。第一个元素为开始，第二个元素为结束，区间运算是包含边界值的，即 xxx>=list.get(0) && xxx<=list.get(1)。

SpEL同样提供了等价的“EQ” 、“NE”、 “GT”、“GE”、 “LT” 、“LE”来表示等于、不等于、大于、大于等于、小于、小于等于，不区分大小写。

四、逻辑表达式：且（and）、或(or)、非(!或NOT)。

1. String expression1 = "2>1 and (!true or !false)";
2. boolean result1 = parser.parseExpression(expression1).getValue(boolean.class);
3. Assert.assertEquals(true, result1);
4. String expression2 = "2>1 and (NOT true or NOT false)";
5. boolean result2 = parser.parseExpression(expression2).getValue(boolean.class);
6. Assert.assertEquals(true, result2);

注：逻辑运算符不支持 **Java**中的 **&&** 和 **||** 。

五、字符串连接及截取表达式：使用“+”进行字符串连接，使用“'String'[0] [index]”来截取一个字符，目前只支持截取一个，如“'Hello ' + 'World!'”得到“Hello World!”；而“'Hello World!'[0]”将返回“H”。

六、三目运算及**Elivis**运算表达式：

三目运算符 **“**表达式**1?**表达式**2:**表达式**3”**用于构造三目运算表达式，如“2>1?true:false”将返回true；

Elivis运算符**“**表达式**1?:**表达式**2”**从Groovy语言引入用于简化三目运算符的，当表达式1为非null时则返回表达式1，当表达式1为null时则返回表达式2，简化了三目运算符方式“表达式1?表达式1:表达式2”，如“null?:false”将返回false，而“true?:false”将返回true；

七、正则表达式：使用“str matches regex，如“'123' matches '\d{3}'”将返回true；

八、括号优先级表达式：使用“(表达式)”构造，括号里的具有高优先级。

5.3.3 类相关表达式

一、类类型表达式：使用“T(Type)”来表示java.lang.Class实例，“Type”必须是类全限定名，“java.lang”包除外，即该包下的类可以不指定包名；使用类类型表达式还可以进行访问类静态方法及类静态字段。

具体使用方法如下：

```
1. @Test
2. public void testClassTypeExpression() {
3. ExpressionParser parser = new SpelExpressionParser();
4. //java.lang包类访问
5. Class<String> result1 = parser.parseExpression("T(String)").getValue(Class.class);
6. Assert.assertEquals(String.class, result1);
7. //其他包类访问
8. String expression2 = "T(cn.javass.spring.chapter5.SpELTest)";
9. Class<String> result2 = parser.parseExpression(expression2).getValue(Class.class);
   Assert.assertEquals(SpELTest.class, result2);
10. //类静态字段访问
11. int result3=parser.parseExpression("T(Integer).MAX_VALUE").getValue(int.class);
12. Assert.assertEquals(Integer.MAX_VALUE, result3);
13. //类静态方法调用
14. int result4 = parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);
15. Assert.assertEquals(1, result4);
16. }
```

对于java.lang包里的可以直接使用“T(String)”访问；其他包必须是类全限定名；可以进行静态字段访问如“T(Integer).MAX_VALUE”；也可以进行静态方法访问如“T(Integer).parseInt('1')”。

二、类实例化：类实例化同样使用java关键字“new”，类名必须是全限定名，但java.lang包内的类型除外，如String、Integer。

```
1. @Test
2. public void testConstructorExpression() {
3. ExpressionParser parser = new SpelExpressionParser();
4. String result1 = parser.parseExpression("new String('haha')").getValue(String.class);
5. Assert.assertEquals("haha", result1);
6. Date result2 = parser.parseExpression("new java.util.Date()").getValue(Date.class);
7. Assert.assertNotNull(result2);
8. }
```

实例化完全跟Java内方式一样。

三、**instanceof**表达式：SpEL支持instanceof运算符，跟Java内使用同义；如“'haha' instanceof T(String)”将返回true。

四、变量定义及引用：变量定义通过EvaluationContext接口的setVariable(variableName, value)方法定义；在表达式中使用“#variableName”引用；除了引用自定义变量，SpE还允许引用根对象及当前上下文对象，使用“#root”引用根对象，使用“#this”引用当前上下文对象；

```
1. @Test
2. public void testVariableExpression() {
3. ExpressionParser parser = new SpelExpressionParser();
4. EvaluationContext context = new StandardEvaluationContext();
5. context.setVariable("variable", "haha");
6. context.setVariable("variable", "haha");
7. String result1 = parser.parseExpression("#variable").getValue(context, String.class);
8. Assert.assertEquals("haha", result1);

9. context = new StandardEvaluationContext("haha");

10. String result2 = parser.parseExpression("#root").getValue(context, String.class);
11. Assert.assertEquals("haha", result2);
12. String result3 = parser.parseExpression("#this").getValue(context, String.class);
13. Assert.assertEquals("haha", result3);
14. }
```

使用“#variable”来引用在EvaluationContext定义的变量；除了可以引用自定义变量，还可以使用“#root”引用根对象，“#this”引用当前上下文对象，此处“#this”即根对象。

五、自定义函数：目前只支持类静态方法注册为自定义函数；SpEL使用StandardEvaluationContext的registerFunction方法进行注册自定义函数，其实完全可以使用setVariable代替，两者其实本质是一样的；

```
1. @Test
2. public void testFunctionExpression() throws SecurityException,
   NoSuchMethodException {
3. ExpressionParser parser = new SpelExpressionParser();
4. StandardEvaluationContext context = new StandardEvaluationContext();
5. Method parseInt = Integer.class.getDeclaredMethod("parseInt", String.class);
6. context.registerFunction("parseInt", parseInt);
7. context.setVariable("parseInt2", parseInt);
8. String expression1 = "#parseInt('3') == #parseInt2('3')";
9. boolean result1 = parser.parseExpression(expression1).getValue(context,
   boolean.class);
10. Assert.assertEquals(true, result1);
11. }
```

此处可以看出“registerFunction”和“setVariable”都可以注册自定义函数，但是两个方法的含义不一样，推荐使用“registerFunction”方法注册自定义函数。

六、赋值表达式：SpEL即允许给自定义变量赋值，也允许给跟对象赋值，直接使用“#variableName=value”即可赋值：

```
1. @Test
2. public void testAssignExpression() {
3. ExpressionParser parser = new SpelExpressionParser();
4. //1.给root对象赋值
5. EvaluationContext context = new StandardEvaluationContext("aaaa");
6. String result1 = parser.parseExpression("#root='aaaaa'").getValue(context,
   String.class);
7. Assert.assertEquals("aaaaa", result1);
8. String result2 = parser.parseExpression("#this='aaaa'").getValue(context, String.class);
9. Assert.assertEquals("aaaa", result2);
10. //2.给自定义变量赋值
11. context.setVariable("#variable", "variable");
12. String result3 = parser.parseExpression("#variable=#root").getValue(context,
   String.class);
13. Assert.assertEquals("aaaa", result3);
14. }
```

使用“#root='aaaaa'”给根对象赋值，使用“"#this='aaaa'”给当前上下文对象赋值，使用“#variable=#root”给自定义变量赋值，很简单。

七、对象属性存取及安全导航表达式：对象属性获取非常简单，即使用如“a.property.property”这种点缀式获取，SpEL对于属性名首字母是不区分大小写的；SpEL还引入了Groovy语言中的安全导航运算符“(对象|属性)?.属性”，用来避免但“?.”前边的表达式为null时抛出空指针异常，而是返回null；修改对象属性值则可以通过赋值表达式或Expression接口的setValue方法修改。

```
1. ExpressionParser parser = new SpelExpressionParser();
2. //1.访问root对象属性
3. Date date = new Date();
4. StandardEvaluationContext context = new StandardEvaluationContext(date);
5. int result1 = parser.parseExpression("Year").getValue(context, int.class);
6. Assert.assertEquals(date.getYear(), result1);
7. int result2 = parser.parseExpression("year").getValue(context, int.class);
8. Assert.assertEquals(date.getYear(), result2);
```

对于当前上下文对象属性及方法访问，可以直接使用属性或方法名访问，比如此处根对象date属性“year”，注意此处属性名首字母不区分大小写。

```
1. //2.安全访问
2. context.setRootObject(null);
3. Object result3 = parser.parseExpression("#root?.year").getValue(context, Object.class);
4. Assert.assertEquals(null, result3);
```

SpEL引入了Groovy的安全导航运算符，比如此处根对象为null，所以如果访问其属性时肯定抛出空指针异常，而采用“?.”安全访问导航运算符将不抛空指针异常，而是简单的返回null。

```
1. //3.给root对象属性赋值
2. context.setRootObject(date);
3. int result4 = parser.parseExpression("Year = 4").getValue(context, int.class);
4. Assert.assertEquals(4, result4);
5. parser.parseExpression("Year").setValue(context, 5);
6. int result5 = parser.parseExpression("Year").getValue(context, int.class);
7. Assert.assertEquals(5, result5);
```

给对象属性赋值可以采用赋值表达式或Expression接口的setValue方法赋值，而且也可以采用点缀方式赋值。

八、对象方法调用：对象方法调用更简单，跟Java语法一样；如“'haha'.substring(2,4)”将返回“ha”；而对于根对象可以直接调用方法；

```
1. Date date = new Date();
2. StandardEvaluationContext context = new StandardEvaluationContext(date);
3. int result2 = parser.parseExpression("getYear()").getValue(context, int.class);
4. Assert.assertEquals(date.getYear(), result2);
```

比如根对象date方法“getYear”可以直接调用。

九、**Bean**引用：SpEL支持使用“@”符号来引用Bean，在引用Bean时需要使用BeanResolver接口实现来查找Bean，Spring提供BeanFactoryResolver实现；

```
1. @Test
2. public void testBeanExpression() {
3. ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext();
4. ctx.refresh();
5. ExpressionParser parser = new SpelExpressionParser();
6. StandardEvaluationContext context = new StandardEvaluationContext();
7. context.setBeanResolver(new BeanFactoryResolver(ctx));
8. Properties result1 = parser.parseExpression("@systemProperties").getValue(context,
   Properties.class);
9. Assert.assertEquals(System.getProperties(), result1);
10. }
```

在示例中我们首先初始化了一个IoC容器，ClassPathXmlApplicationContext 实现默认会把“System.getProperties()”注册为“systemProperties”Bean，因此我们使用“@systemProperties”来引用该Bean。

5.3.3 集合相关表达式

一、内联**List**：从Spring3.0.4开始支持内联List，使用{表达式，……}定义内联List，如“{1,2,3}”将返回一个整型的ArrayList，而“{}”将返回空的List，对于字面量表达式列表，SpEL会使用java.util.Collections.unmodifiableList方法将列表设置为不可修改。

```
1. //将返回不可修改的空List
2. List<Integer> result2 = parser.parseExpression("{}").getValue(List.class);
```

```
1. //对于字面量列表也将返回不可修改的List
2. List<Integer> result1 = parser.parseExpression("{1,2,3}").getValue(List.class);
3. Assert.assertEquals(new Integer(1), result1.get(0));
4. try {
5. result1.set(0, 2);
6. //不可能执行到这，对于字面量列表不可修改
7. Assert.fail();
8. } catch (Exception e) {
9. }
```

```
1. //对于列表中只要有一个不是字面量表达式，将只返回原始List，
2. //不会进行不可修改处理
3. String expression3 = "{{1+2,2+4},{3,4+4}}";
4. List<List<Integer>> result3 = parser.parseExpression(expression3).getValue(List.class);
5. result3.get(0).set(0, 1);
6. Assert.assertEquals(2, result3.size());
```

```
1. //声明一个大小为2的一维数组并初始化
2. int[] result2 = parser.parseExpression("new int[2]{1,2}").getValue(int[].class);
```

```
1. //定义一维数组但不初始化
2. int[] result1 = parser.parseExpression("new int[1]").getValue(int[].class);
```

二、内联数组：和Java 数组定义类似，只是在定义时进行多维数组初始化。

```
1. //定义多维数组但不初始化
2. int[][][] result3 = parser.parseExpression("new int[1][2][3]").getValue(int[][][].class);
```

```
1. //错误的定义多维数组，多维数组不能初始化
2. String expression4 = "new int[1][2][3]{{1}{2}{3}}";
3. try {
4. int[][][] result4 = parser.parseExpression(expression4).getValue(int[][][].class);
5. Assert.fail();
6. } catch (Exception e) {
7. }
```

三、集合，字典元素访问：SpEL目前支持所有集合类型和字典类型的元素访问，使用“集合[索引]”访问集合元素，使用“map[key]”访问字典元素；

```
1. //SpEL内联List访问
2. int result1 = parser.parseExpression("{1,2,3}[0]").getValue(int.class);
3. //即list.get(0)
4. Assert.assertEquals(1, result1);
```

```
1. //SpEL目前支持所有集合类型的访问
2. Collection<Integer> collection = new HashSet<Integer>();
3. collection.add(1);
4. collection.add(2);
5. EvaluationContext context2 = new StandardEvaluationContext();
6. context2.setVariable("collection", collection);
7. int result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);
8. //对于任何集合类型通过Iterator来定位元素
9. Assert.assertEquals(2, result2);
```

```
1. //SpEL对Map字典元素访问的支持
2. Map<String, Integer> map = new HashMap<String, Integer>();
3. map.put("a", 1);
4. EvaluationContext context3 = new StandardEvaluationContext();
5. context3.setVariable("map", map);
6. int result3 = parser.parseExpression("#map['a']").getValue(context3, int.class);
7. Assert.assertEquals(1, result3);
```

注：集合元素访问是通过**Iterator**遍历来定位元素位置的。

四、列表，字典，数组元素修改：可以使用赋值表达式或Expression接口的setValue方法修改；

```
1. //1.修改数组元素值
2. int[] array = new int[] {1, 2};
3. EvaluationContext context1 = new StandardEvaluationContext();
4. context1.setVariable("array", array);
5. int result1 = parser.parseExpression("#array[1] = 3").getValue(context1, int.class);
6. Assert.assertEquals(3, result1);
```

```
1. //2.修改集合值
2. Collection<Integer> collection = new ArrayList<Integer>();
3. collection.add(1);
4. collection.add(2);
5. EvaluationContext context2 = new StandardEvaluationContext();
6. context2.setVariable("collection", collection);
```

```
7. int result2 = parser.parseExpression("#collection[1] = 3").getValue(context2, int.class);
8. Assert.assertEquals(3, result2);
9. parser.parseExpression("#collection[1]").setValue(context2, 4);
10. result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);
11. Assert.assertEquals(4, result2);
```

```
1. //3.修改map元素值
2. Map<String, Integer> map = new HashMap<String, Integer>();
3. map.put("a", 1);
4. EvaluationContext context3 = new StandardEvaluationContext();
5. context3.setVariable("map", map);
6. int result3 = parser.parseExpression("#map['a'] = 2").getValue(context3, int.class);
7. Assert.assertEquals(2, result3);
```

对数组修改直接对“#array[index]”赋值即可修改元素值，同理适用于集合和字典类型。

五、集合投影：在SQL中投影指从表中选择出列，而在SpEL指根据集合中的元素中通过选择来构造另一个集合，该集合和原集合具有相同数量的元素；SpEL使用“（list|map）.![投影表达式]”来进行投影运算：

```
1. //1.首先准备测试数据
2. Collection<Integer> collection = new ArrayList<Integer>();
3. collection.add(4); collection.add(5);
4. Map<String, Integer> map = new HashMap<String, Integer>();
5. map.put("a", 1); map.put("b", 2);
```

```
1. //2.测试集合或数组
2. EvaluationContext context1 = new StandardEvaluationContext();
3. context1.setVariable("collection", collection);
4. Collection<Integer> result1 =
5. parser.parseExpression("#collection.![#this+1]").getValue(context1, Collection.class);
6. Assert.assertEquals(2, result1.size());
7. Assert.assertEquals(new Integer(5), result1.iterator().next());
```

对于集合或数组使用如上表达式进行投影运算，其中投影表达式中“#this”代表每个集合或数组元素，可以使用比如“#this.property”来获取集合元素的属性，其中“#this”可以省略。

```
1. //3.测试字典
2. EvaluationContext context2 = new StandardEvaluationContext();
3. context2.setVariable("map", map);
4. List<Integer> result2 =
5. parser.parseExpression("#map.![ value+1]").getValue(context2, List.class);
6. Assert.assertEquals(2, result2.size());
```

SpEL投影运算还支持Map投影，但Map投影最终只能得到List结果，如上所示，对于投影表达式中的“#this”将是Map.Entry，所以可以使用“value”来获取值，使用“key”来获取键。

六、集合选择：在SQL中指使用select进行选择行数据，而在SpEL指根据原集合通过条件表达式选择出满足条件的元素并构造为新的集合，SpEL使用“(list|map).?[选择表达式]”，其中选择表达式结果必须是boolean类型，如果true则选择的元素将添加到新集合中，false将不添加到新集合中。

```
1. //1.首先准备测试数据
2. Collection<Integer> collection = new ArrayList<Integer>();
3. collection.add(4); collection.add(5);
4. Map<String, Integer> map = new HashMap<String, Integer>();
5. map.put("a", 1); map.put("b", 2);
```

```
1. //2.集合或数组测试
2. EvaluationContext context1 = new StandardEvaluationContext();
3. context1.setVariable("collection", collection);
4. Collection<Integer> result1 =
5. parser.parseExpression("#collection.?[#this>4]").getValue(context1, Collection.class);
6. Assert.assertEquals(1, result1.size());
7. Assert.assertEquals(new Integer(5), result1.iterator().next());
```

对于集合或数组选择，如“#collection.?[#this>4]”将选择出集合元素值大于4的所有元素。选择表达式必须返回布尔类型，使用“#this”表示当前元素。

```
1. //3.字典测试
2. EvaluationContext context2 = new StandardEvaluationContext();
3. context2.setVariable("map", map);
4. Map<String, Integer> result2 =
5. parser.parseExpression("#map.?[#this.key != 'a']").getValue(context2, Map.class);
6. Assert.assertEquals(1, result2.size());
7. List<Integer> result3 =
8. parser.parseExpression("#map.?[key != 'a'].![value+1]").getValue(context2, List.class);
9. Assert.assertEquals(new Integer(3), result3.iterator().next());
```

对于字典选择，如“#map.?[#this.key != 'a']”将选择键值不等于”a”的，其中选择表达式中“#this”是Map.Entry类型，而最终结果还是Map，这点和投影不同；集合选择和投影可以一起使用，如“#map.?[key != 'a'].![value+1]”将首先选择键值不等于”a”的，然后在选出的Map中再进行“value+1”的投影。

5.3.4 表达式模板

模板表达式就是由字面量与一个或多个表达式块组成。每个表达式块由“前缀+表达式+后缀”形式组成，如“${1+2}”即表达式块。在前边我们已经介绍了使用ParserContext接口实现来定义表达式是否是模板及前缀和后缀定义。在此就不多介绍了，如“Error ${#v0} ${#v1}”表达式表示由字面量“Error ”、模板表达式“#v0”、模板表达式“#v1”组成，其中v0和v1表示自定义变量，需要在上下文定义。

原创内容 转自请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/2463.html】

【第五章】Spring表达式语言 之 5.4在Bean定义中使用EL—跟我学spring3

5.4.1 xml风格的配置

SpEL支持在Bean定义时注入，默认使用“#{SpEL表达式}”表示，其中“#root”根对象默认可以认为是ApplicationContext，只有ApplicationContext实现默认支持SpEL，获取根对象属性其实是获取容器中的Bean。

首先看下配置方式（chapter5/el1.xml）吧：

```
1.  <bean id="world" class="java.lang.String">
2.  <constructor-arg value="#{' World!'}"/>
3.  </bean>
4.  <bean id="hello1" class="java.lang.String">
5.  <constructor-arg value="#{'Hello'}#{world}"/>
6.  </bean>
7.  <bean id="hello2" class="java.lang.String">
8.  <constructor-arg value="#{'Hello' + world}"/>
9.  <!-- 不支持嵌套的 -->
10. <!--<constructor-arg value="#{'Hello'#{world}}"/>-->
11. </bean>
12. <bean id="hello3" class="java.lang.String">
13. <constructor-arg value="#{'Hello' + @world}"/>
14. </bean>
```

模板默认以前缀“#{”开头，以后缀“}”结尾，且不允许嵌套，如“#{'Hello'#{world}}”错误，如“#{'Hello' + world}”中“world”默认解析为Bean。当然可以使用“@bean”引用了。

接下来测试一下吧：

```
1.  @Test
2.  public void testXmlExpression() {
3.  ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter5/el1.xml");
4.  String hello1 = ctx.getBean("hello1", String.class);
5.  String hello2 = ctx.getBean("hello2", String.class);
6.  String hello3 = ctx.getBean("hello3", String.class);
7.  Assert.assertEquals("Hello World!", hello1);
8.  Assert.assertEquals("Hello World!", hello2);
9.  Assert.assertEquals("Hello World!", hello3);
```

10. }

是不是很简单，除了XML配置方式，Spring还提供一种注解方式@Value，接着往下看吧。

5.4.2 注解风格的配置

基于注解风格的SpEL配置也非常简单，使用@Value注解来指定SpEL表达式，该注解可以放到字段、方法及方法参数上。

测试Bean类如下，使用@Value来指定SpEL表达式：

```
1.  package cn.javass.spring.chapter5;
2.  import org.springframework.beans.factory.annotation.Value;
3.  public class SpELBean {
4.  @Value("#{'Hello' + world}")
5.  private String value;
6.  //setter和getter由于篇幅省略，自己写上
7.  }
```

首先看下配置文件(chapter5/el2.xml)：

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.  xmlns:context="http://www.springframework.org/schema/context"
5.  xsi:schemaLocation="
6.  http://www.springframework.org/schema/beans
7.  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8.  http://www.springframework.org/schema/context
9.  http://www.springframework.org/schema/context/spring-context-3.0.xsd"&gt;
10. <context:annotation-config/>
11. <bean id="world" class="java.lang.String">
12. <constructor-arg value="#{' World!'}"/>
13. </bean>
14. <bean id="helloBean1" class="cn.javass.spring.chapter5.SpELBean"/>
15. <bean id="helloBean2" class="cn.javass.spring.chapter5.SpELBean">
16. <property name="value" value="haha"/>
17. </bean>
18. </beans>
```

配置时必须使用“**<context:annotation-config/>**”来开启对注解的支持。

有了配置文件那开始测试吧：

```
1. @Test
2. public void testAnnotationExpression() {
3. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter5/el2.xml");
4. SpELBean helloBean1 = ctx.getBean("helloBean1", SpELBean.class);
5. Assert.assertEquals("Hello World!", helloBean1.getValue());
6. SpELBean helloBean2 = ctx.getBean("helloBean2", SpELBean.class);
7. Assert.assertEquals("haha", helloBean2.getValue());
8. }
```

其中“helloBean1 ”值是SpEL表达式的值，而“helloBean2”是通过setter注入的值，这说明setter注入将覆盖@Value的值。

5.4.3 在Bean定义中SpEL的问题

如果有同学问“#{我不是SpEL表达式}”不是SpEL表达式，而是公司内部的模板，想换个前缀和后缀该如何实现呢？

那我们来看下Spring如何在IoC容器内使用BeanExpressionResolver接口实现来求值SpEL表达式，那如果我们通过某种方式获取该接口实现，然后把前缀后缀修改了不就可以了。

此处我们使用BeanFactoryPostProcessor接口提供postProcessBeanFactory回调方法，它是在IoC容器创建好但还未进行任何Bean初始化时被ApplicationContext实现调用，因此在这个阶段把SpEL前缀及后缀修改掉是安全的，具体代码如下：

```
1. package cn.javass.spring.chapter5;
2. import org.springframework.beans.BeansException;
3. import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
4. import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
5. import org.springframework.context.expression.StandardBeanExpressionResolver;
6. public class SpELBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
7. @Override
8. public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
9. throws BeansException {
10. StandardBeanExpressionResolver resolver = (StandardBeanExpressionResolver)
    beanFactory.getBeanExpressionResolver();
11. resolver.setExpressionPrefix("%{");
12. resolver.setExpressionSuffix("}");
13. }
14. }
```

首先通过 ConfigurableListableBeanFactory的getBeanExpressionResolver方法获取BeanExpressionResolver实现，其次强制类型转换为StandardBeanExpressionResolver，其为Spring默认实现，然后改掉前缀及后缀。

开始测试吧，首先准备配置文件(chapter5/el3.xml)：

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xsi:schemaLocation="
6. http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8. http://www.springframework.org/schema/context
9. http://www.springframework.org/schema/context/spring-context-3.0.xsd"&gt;
10. <context:annotation-config/>
11. <bean class="cn.javass.spring.chapter5.SpELBeanFactoryPostProcessor"/>
12. <bean id="world" class="java.lang.String">
13. <constructor-arg value="%{' World!'}"/>
14. </bean>
15. <bean id="helloBean1" class="cn.javass.spring.chapter5.SpELBean"/>
16. <bean id="helloBean2" class="cn.javass.spring.chapter5.SpELBean">
17. <property name="value" value="%{'Hello' + world}"/>
18. </bean>
19. </beans>
```

配置文件和注解风格的几乎一样，只有SpEL表达式前缀变为“%{”了，并且注册了“cn.javass.spring.chapter5.SpELBeanFactoryPostProcessor”Bean，用于修改前缀和后缀的。

写测试代码测试一下吧：

```
1. @Test
2. public void testPrefixExpression() {
3. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter5/el3.xml");
4. SpELBean helloBean1 = ctx.getBean("helloBean1", SpELBean.class);
5. Assert.assertEquals("#{'Hello' + world}", helloBean1.getValue());
6. SpELBean helloBean2 = ctx.getBean("helloBean2", SpELBean.class);
7. Assert.assertEquals("Hello World!", helloBean2.getValue());
8. }
```

此处helloBean1 中通过@Value注入的“#{'Hello' + world}”结果还是“#{'Hello' + world}”说明不对其进行SpEL表达式求值了，而helloBean2使用“%{'Hello' + world}”注入，得到正确的“"Hello World!”。

【第六章】 AOP 之 6.1 AOP基础 ——跟我学spring3

6.1.1 AOP是什么

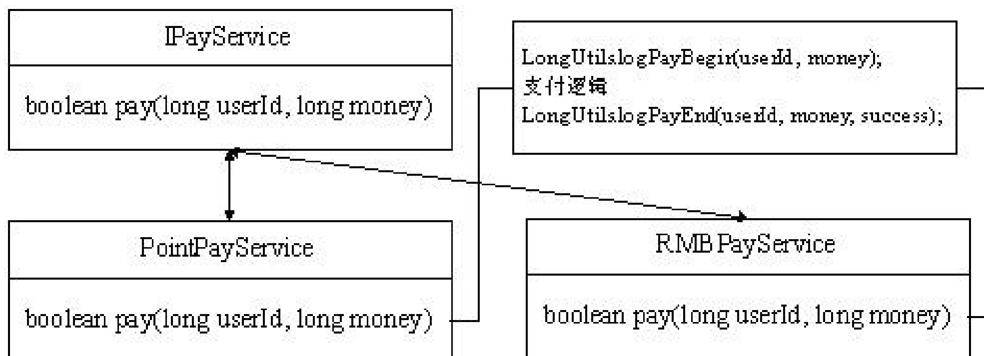
考虑这样一个问题：需要对系统中的某些业务做日志记录，比如支付系统中的支付业务需要记录支付相关日志，对于支付系统可能相当复杂，比如可能有自己的支付系统，也可能引入第三方支付平台，面对这样的支付系统该如何解决呢？

- 传统解决方案：

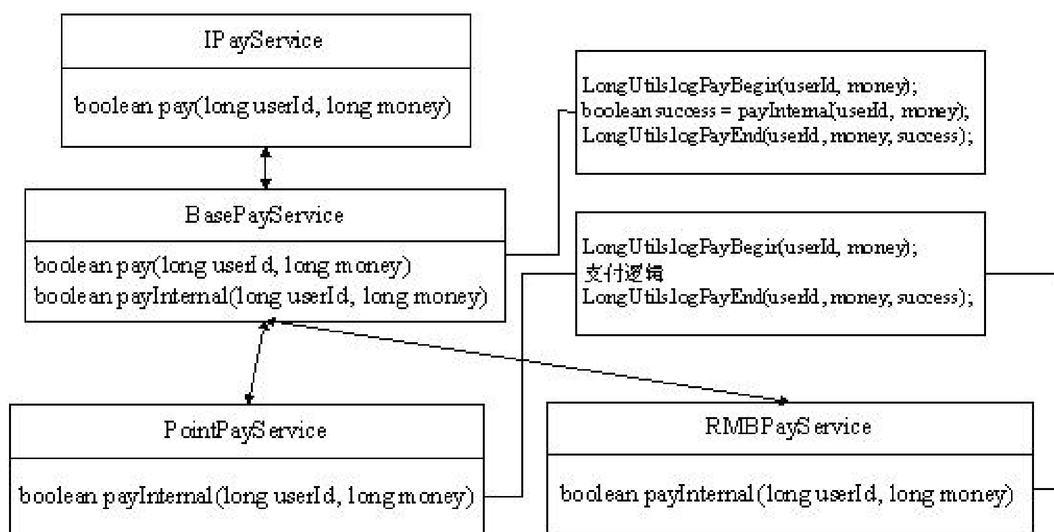
1）日志部分提前公共类LogUtils，定义“longPayBegin”方法用于记录支付开始日志，“logPayEnd”用于记录支付结果：

LogUtils
logPayBegin(long userId, long money) logPayEnd(long userId, long money, boolean success)

2）支付部分，定义IPayService接口并定义支付方法“pay”，并定义了两个实现：“PointPayService”表示积分支付，“RMBPayService”表示人民币支付；并且在每个支付实现中支付逻辑和记录日志：



3）支付实现很明显有重复代码，这个重复很明显可以使用模板设计模式消除重复：

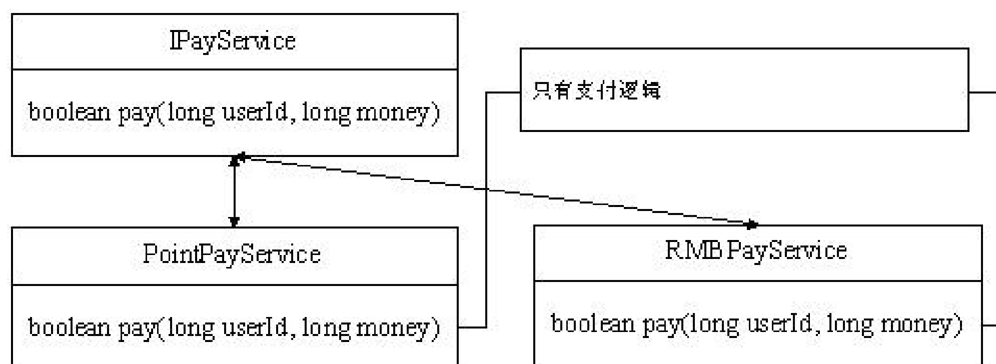


4）到此我们设计了一个可以复用的接口；但大家觉得这样记录日志会很好吗，有没有更好的解决方案？

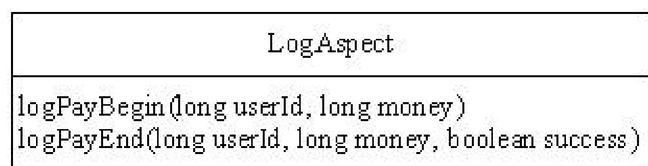
如果对积分支付方式添加统计功能，比如在支付时记录下用户总积分数、当前消费的积分数，那我们该如何做呢？直接修改源代码添加日志记录，这完全违背了面向对象最重要的原则之一：开闭原则（对扩展开放，对修改关闭）？

- 更好的解决方案：在我们的支付组件中由于使用了日志组件，即日志模块横切于支付组件，在传统程序设计中很难将日志组件分离出来，即不耦合我们的支付组件；因此面向方面编程AOP就诞生了，它能分离我们的组件，使组件完全不耦合:

1）采用面向方面编程后，我们的支付组件看起来如下所示，代码中不再有日志组件的任何东西；



2）所以日志相关的提取到一个切面中，AOP实现者会在合适的时候将日志功能织入到我们的支付组件中去，从而完全解耦支付组件和日志组件。



看到这大家可能不是很理解，没关系，先往下看。

> |
>
> 面向方面编程(AOP)：也可称为面向切面编程，是一种编程范式，提供从另一个角度来考虑程序结构从而完善面向对象编程(OOP)。
>
> |

在进行OOP开发时，都是基于对组件（比如类）进行开发，然后对组件进行组合，OOP最大问题就是无法解耦组件进行开发，比如我们上边举例，而AOP就是为了克服这个问题而出现的，它来进行这种耦合的分离。

AOP为开发者提供一种进行横切关注点（比如日志关注点横切了支付关注点）分离并织入的机制，把横切关注点分离，然后通过某种技术织入到系统中，从而无耦合的完成了我们的功能。

6.1.2 能干什么

AOP主要用于横切关注点分离和织入，因此需要理解横切关注点和织入：

- 关注点：可以认为是所关注的任何东西，比如上边的支付组件；
- 关注点分离：将问题细化从而单独部分，即可以理解为不可再分割的组件，如上边的日志组件和支付组件；
- 横切关注点：一个组件无法完成需要的功能，需要其他组件协作完成，如日志组件横切于支付组件；
- 织入：横切关注点分离后，需要通过某种技术将横切关注点融合到系统中从而完成需要的功能，因此需要织入，织入可能在编译期、加载期、运行期等进行。

横切关注点可能包含很多，比如非业务的：日志、事务处理、缓存、性能统计、权限控制等等这些非业务的基础功能；还可能是业务的：如某个业务组件横切于多个模块。如图6-1

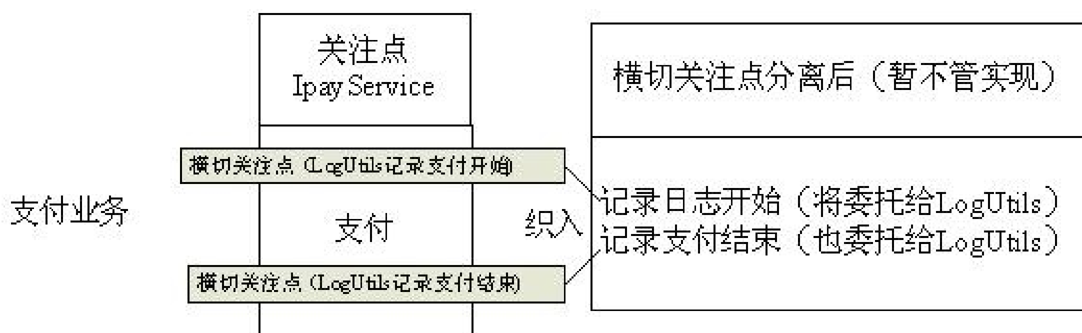


图6-1 关注点与横切关注点

传统支付形式，流水方式：



面向切面方式，先将横切关注点分离，再将横切关注点织入到支付系统中：



AOP能干什么：

- 用于横切关注点的分离和织入横切关注点到系统；比如上边提到的日志等等；
- 完善OOP；
- 降低组件和模块之间的耦合性；
- 使系统容易扩展；
- 而且由于关注点分离从而可以获得组件的更好复用。

6.1.3 AOP的基本概念

在进行AOP开发前，先熟悉几个概念：

- 连接点（**Jointpoint**）：表示需要在程序中插入横切关注点的扩展点，连接点可能是类初始化、方法执行、方法调用、字段调用或处理异常等等，Spring只支持方法执行连接点，在**AOP**中表示为**“**在哪里干**”**；
- 切入点（**Pointcut**）：选择一组相关连接点的模式，即可以认为连接点的集合，Spring支持perl5正则表达式和AspectJ切入点模式，Spring默认使用AspectJ语法，在**AOP**中表示为**“**在哪里干的集合**”**；
- 通知（**Advice**）：在连接点上执行的行为，通知提供了在AOP中需要在切入点所选择的连接点处进行扩展现有行为的手段；包括前置通知（before advice）、后置通知(after advice)、环绕通知（around advice），在Spring中通过代理模式实现AOP，并通过拦截器模式以环绕连接点的拦截器链织入通知；在**AOP**中表示为**“**干什么**”**；
- 方面/切面（**Aspect**）：横切关注点的模块化，比如上边提到的日志组件。可以认为是通知、引入和切入点的组合；在Spring中可以使用Schema和@AspectJ方式进行组织实现；在**AOP**中表示为**“**在哪干和干什么集合**”**；

- 引入（**inter-type declaration**）：也称为内部类型声明，为已有的类添加额外新的字段或方法，Spring允许引入新的接口（必须对应一个实现）到所有被代理对象（目标对象）,在**AOP**中表示为**“**干什么（引入什么）**”**；
- 目标对象（**Target Object**）：需要被织入横切关注点的对象，即该对象是切入点选择的对象，需要被通知的对象，从而也可称为“被通知对象”；由于Spring AOP 通过代理模式实现，从而这个对象永远是被代理对象，在**AOP**中表示为**“**对谁干**”**；
- **AOP**代理（**AOP Proxy**）：AOP框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是通过代理来对目标对象应用切面。在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。
- 织入（**Weaving**）：织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程，织入可以在编译期、类装载期、运行期进行。

在AOP中，通过切入点选择目标对象的连接点，然后在目标对象的相应连接点处织入通知，而切入点和通知就是切面（横切关注点），而在目标对象连接点处应用切面的实现方式是通过AOP代理对象，如图6-2所示。

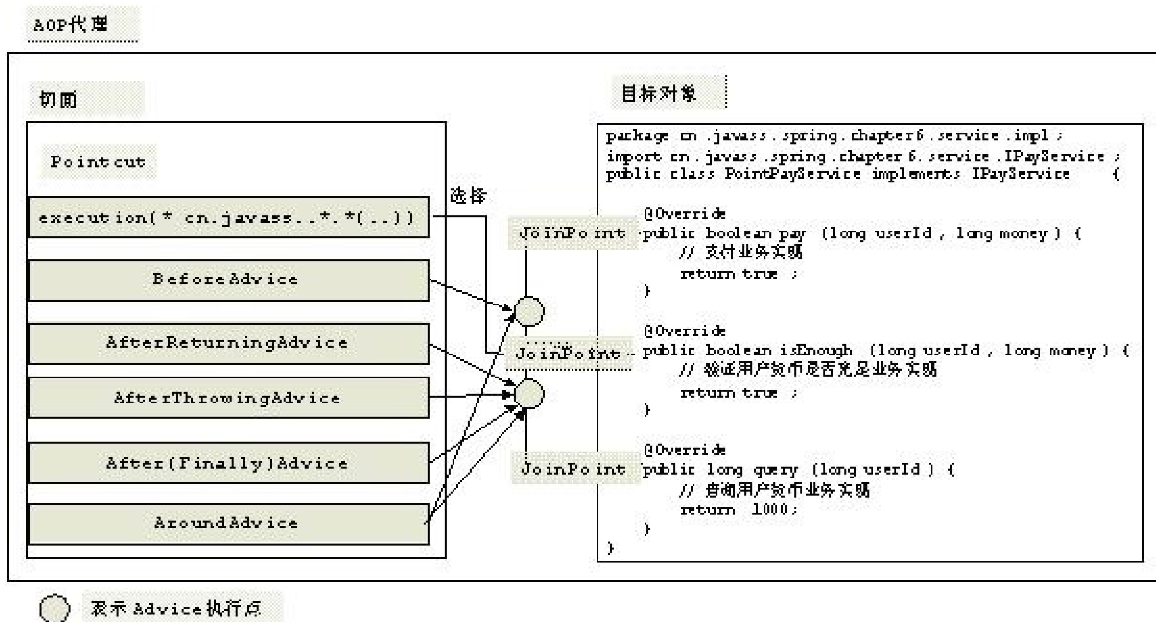


图6-2 概念关系

接下来再让我们具体看看Spring有哪些通知类型：

- 前置通知（**Before Advice**）**:**在切入点选择的连接点处的方法之前执行的通知，该通知不影响正常程序执行流程（除非该通知抛出异常，该异常将中断当前方法链的执行而返回）。
- 后置通知（**After Advice**）**:**在切入点选择的连接点处的方法之后执行的通知，包括如下类型的后置通知：
    - 后置返回通知（**After returning Advice**）**:**在切入点选择的连接点处的方法正常执行完毕时执行的通知，必须是连接点处的方法没抛出任何异常正常返回时才调用后置通知。
    - 后置异常通知（**After throwing Advice**）**:** 在切入点选择的连接点处的方法抛出异

常返回时执行的通知，必须是连接点处的方法抛出任何异常返回时才调用异常通知。

  - 后置最终通知（**After finally Advice**）：在切入点选择的连接点处的方法返回时执行的通知，不管抛没抛出异常都执行，类似于Java中的finally块。

- 环绕通知（**Around Advices**）：环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知可以在方法调用之前和之后自定义任何行为，并且可以决定是否执行连接点处的方法、替换返回值、抛出异常等等。

各种通知类型在UML序列图中的位置如图6-3所示：

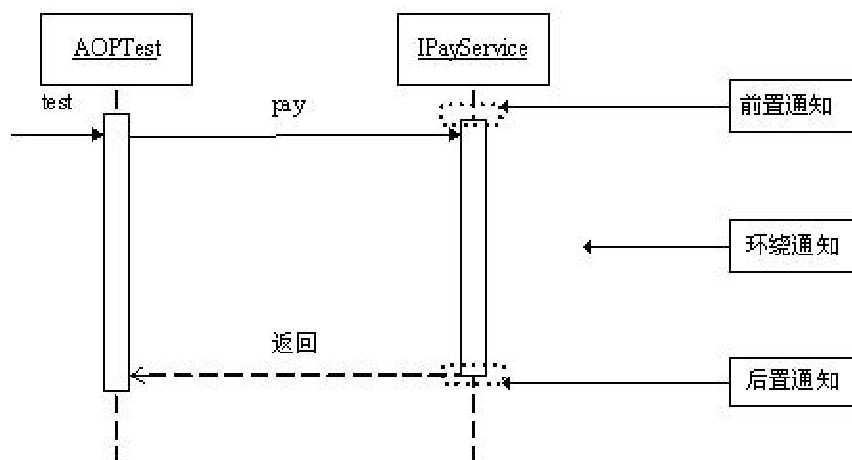


图6-3 通知类型

6.1.4 AOP代理

AOP代理就是AOP框架通过代理模式创建的对象，Spring使用JDK动态代理或CGLIB代理来实现，Spring缺省使用JDK动态代理来实现，从而任何接口都可别代理，如果被代理的对象实现不是接口将默认使用CGLIB代理，不过CGLIB代理当然也可应用到接口。

AOP代理的目的就是将切面织入到目标对象。

概念都将完了，接下来让我们看一下AOP的 HelloWorld!吧。

原创内容 转自请注明出处【http://sishuok.com/forum/blogPost/list/2466.html】

【第六章】 AOP 之 6.2 AOP的HelloWorld ——跟我学spring3

6.2.1 准备环境

首先准备开发需要的jar包，请到spring-framework-3.0.5.RELEASE-dependencies.zip和spring-framework-3.0.5.RELEASE-with-docs中查找如下jar包：

```
org.springframework.aop-3.0.5.RELEASE.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

com.springsource.net.sf.cglib-2.2.0.jar
```

将这些jar包添加到"Build Path"下。

6.2.2 定义目标类

1）定义目标接口：

1. package cn.javass.spring.chapter6.service;
2. public interface IHelloWorldService {
3. public void sayHello();
4. }

2）定义目标接口实现：

1. package cn.javass.spring.chapter6.service.impl;
2. import cn.javass.spring.chapter6.service.IHelloWorldService;
3. public class HelloWorldService implements IHelloWorldService {
4. @Override
5. public void sayHello() {
6. System.out.println("============Hello World!");
7. }
8. }

注：在日常开发中最后将业务逻辑定义在一个专门的service包下，而实现定义在service包下的impl包中，服务接口以IXXXService形式，而服务实现就是XXXService，这就是规约设计，见名知义。当然可以使用公司内部更好的形式，只要大家都好理解就可以了。

6.2.2 定义切面支持类

有了目标类，该定义切面了，切面就是通知和切入点的组合，而切面是通过配置方式定义的，因此这定义切面前，我们需要定义切面支持类，切面支持类提供了通知实现：

```
1. package cn.javass.spring.chapter6.aop;
2. public class HelloWorldAspect {
3. //前置通知
4. public void beforeAdvice() {
5. System.out.println("===========before advice");
6. }
7. //后置最终通知
8. public void afterFinallyAdvice() {
9. System.out.println("===========after finally advice");
10. }
11. }
```

此处HelloWorldAspect类不是真正的切面实现，只是定义了通知实现的类，在此我们可以把它看作就是缺少了切入点的切面。

注：对于AOP相关类最后专门放到一个包下，如“aop”包，因为AOP是动态织入的，所以如果某个目标类被AOP拦截了并应用了通知，可能很难发现这个通知实现在哪个包里，因此推荐使用规约命名，方便以后维护人员查找相应的AOP实现。

6.2.3 在XML中进行配置

有了通知实现，那就让我们来配置切面吧：

1）首先配置AOP需要aop命名空间，配置头如下：

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="
6. http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8. http://www.springframework.org/schema/aop
9. http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"&gt;
10. </beans>
```

2）配置目标类：

```
1. <bean id="helloWorldService"
2. class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>
```

3）配置切面：

```
1. <bean id="aspect" class="cn.javass.spring.chapter6.aop.HelloWorldAspect"/>
2. <aop:config>
3. <aop:pointcut id="pointcut" expression="execution( cn.javass...*(..))"/>
4. <aop:aspect ref="aspect">
5. <aop:before pointcut-ref="pointcut" method="beforeAdvice"/>
6. <aop:after pointcut="execution( cn.javass...*(..))" method="afterFinallyAdvice"/>
7. </aop:aspect>
8. </aop:config>
```

切入点使用<aop:config>标签下的<aop:pointcut>配置，expression属性用于定义切入点模式，默认是AspectJ语法，“execution(cn.javass...*(..))”表示匹配cn.javass包及子包下的任何方法执行。

切面使用<aop:config>标签下的<aop:aspect>标签配置，其中“ref”用来引用切面支持类的方法。

前置通知使用<aop:aspect>标签下的<aop:before>标签来定义，pointcut-ref属性用于引用切入点Bean，而method用来引用切面通知实现类中的方法，该方法就是通知实现，即在目标类方法执行之前调用的方法。

最终通知使用<aop:aspect>标签下的<aop:after >标签来定义，切入点除了使用pointcut-ref属性来引用已经存在的切入点，也可以使用pointcut属性来定义，如pointcut="execution(cn.javass...*(..))"，method属性同样是指定通知实现，即在目标类方法执行之后调用的方法。

6.2.4 运行测试

测试类非常简单，调用被代理Bean跟调用普通Bean完全一样，Spring AOP将为目标对象创建AOP代理，具体测试代码如下：

```
1. package cn.javass.spring.chapter6;
2. import org.junit.Test;
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import cn.javass.spring.chapter6.service.IHelloWorldService;
6. import cn.javass.spring.chapter6.service.IPayService;
7. public class AopTest {
8. @Test
9. public void testHelloworld() {
10. ApplicationContext ctx = new
    ClassPathXmlApplicationContext("chapter6/helloworld.xml");
11. IHelloWorldService helloworldService =
```

12. ctx.getBean("helloWorldService", IHelloWorldService.class);
13. helloworldService.sayHello();
14. }
15. }

该测试将输出如下如下内容：

1. ===========before advice
2. ============Hello World!
3. ===========after finally advice

从输出我们可以看出：前置通知在切入点选择的连接点（方法）之前允许，而后置通知将在连接点（方法）之后执行，具体生成AOP代理及执行过程如图6-4所示。

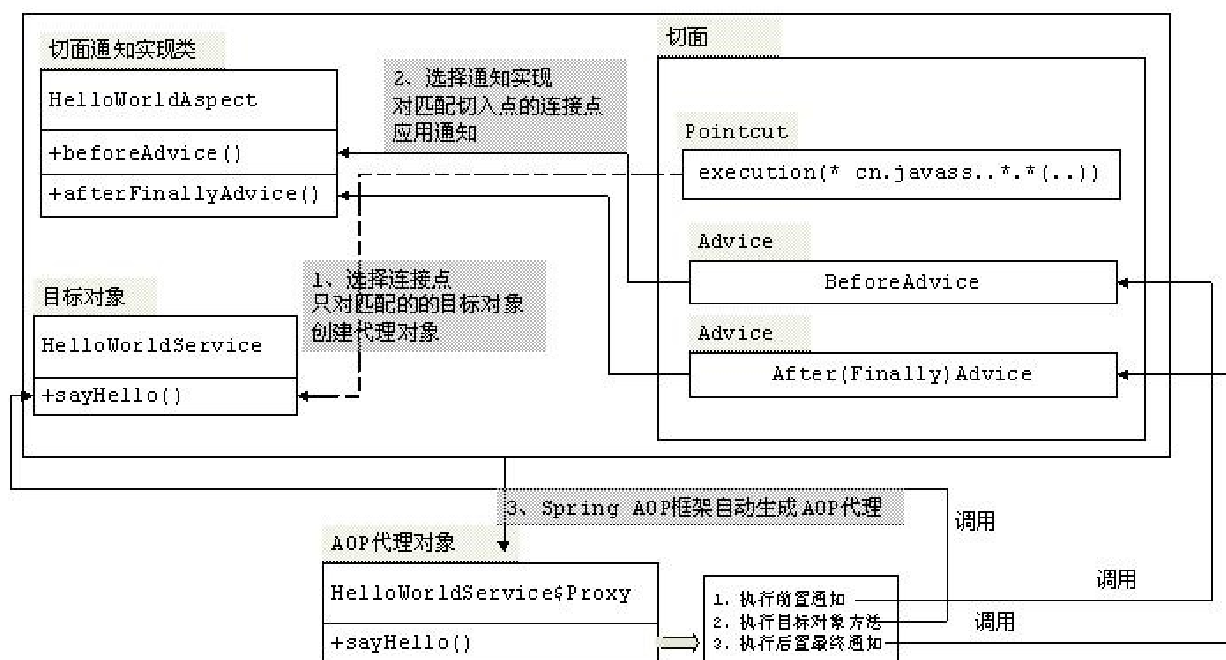


图6-4 Spring AOP框架生成AOP代理过程

原创内容 转自请注明出处【http://sishuok.com/forum/blogPost/list/2467.html】

【第六章】 AOP 之 6.3 基于Schema的AOP ——跟我学spring3

6.3 基于Schema的AOP

基于Schema的AOP从Spring2.0之后通过“aop”命名空间来定义切面、切入点及声明通知。

在Spring配置文件中，所以AOP相关定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，配置顺序不可变。

- <aop:pointcut>：用来定义切入点，该切入点可以重用；
- <aop:advisor>：用来定义只有一个通知和一个切入点的切面；
- <aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和advisor的区别就在此，advisor只包含一个通知和一个切入点。

<aop:config>	AOP定义开始（有序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:advisor/>	Advisor定义（零个或多个）
<aop:aspect>	切面定义开始（零个或多个，无序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:before"/>	前置通知（零个或多个）
<aop:after-returning/>	后置返回通知（零个或多个）
<aop:after-throwing/>	后置异常通知（零个或多个）
<aop:after/>	后置最终通知（零个或多个）
<aop:around/>	环绕通知（零个或多个）
<aop:declare-parents/>	引入定义（零个或多个）
</aop:aspect>	切面定义开始（零个或多个）
</aop:config>	AOP定义结束

6.3.1 声明切面

切面就是包含切入点和通知的对象，在Spring容器中将被定义为一个Bean，Schema方式的切面需要一个切面支持Bean，该支持Bean的字段和方法提供了切面的状态和行为信息，并通过配置方式来指定切入点和通知实现。

切面使用<aop:aspect>标签指定，ref属性用来引用切面支持Bean。

切面支持Bean“aspectSupportBean”跟普通Bean完全一样使用，切面使用“ref”属性引用它。

6.3.2 声明切入点

切入点在Spring中也是一个Bean，Bean定义方式可以有很三种方式：

1）在**<aop:config>**标签下使用**<aop:pointcut>**声明一个切入点**Bean**，该切入点可以被多个切面使用，对于需要共享使用的切入点最好使用该方式，该切入点使用id属性指定Bean名字，在通知定义时使用pointcut-ref属性通过该id引用切入点，expression属性指定切入点表达式：

```
1. <aop:config>
2. <aop:pointcut id="pointcut" expression="execution( cn.javass...*(..))"/>
3. <aop:aspect ref="aspectSupportBean">
4. <aop:before pointcut-ref="pointcut" method="before"/>
5. </aop:aspect>
6. </aop:config>
```

2）在**<aop:aspect>**标签下使用**<aop:pointcut>**声明一个切入点**Bean**，该切入点可以被多个切面使用，但一般该切入点只被该切面使用，当然也可以被其他切面使用，但最好不要那样使用，该切入点使用id属性指定Bean名字，在通知定义时使用pointcut-ref属性通过该id引用切入点，expression属性指定切入点表达式：

```
1. <aop:config>
2. <aop:aspect ref="aspectSupportBean">
3. <aop:pointcut id=" pointcut" expression="execution( cn.javass...*(..))"/>
4. <aop:before pointcut-ref="pointcut" method="before"/>
5. </aop:aspect>
6. </aop:config>
```

3）匿名切入点**Bean**，可以在声明通知时通过pointcut属性指定切入点表达式，该切入点是匿名切入点，只被该通知使用：

```
1. <aop:config>
2. <aop:aspect ref="aspectSupportBean">
3. <aop:after pointcut="execution( cn.javass...*(..))" method="afterFinallyAdvice"/>
4. </aop:aspect>
5. </aop:config>
```

6.3.3 声明通知

基于Schema方式支持前边介绍的5中通知类型：

一、前置通知：在切入点选择的方法之前执行，通过<aop:aspect>标签下的<aop:before>标签声明：

```
1. <aop:before pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. method="前置通知实现方法名"
3. arg-names="前置通知实现方法参数列表参数名字"/>
```

pointcut和**pointcut-ref**：二者选一，指定切入点；

method：指定前置通知实现方法名，如果是多态需要加上参数类型，多个用“，”隔开，如beforeAdvice(java.lang.String)；

arg-names：指定通知实现方法的参数名字，多个用“，”分隔，可选，类似于【3.1.2 构造器注入】中的参数名注入限制：在**class**文件中没生成变量调试信息是获取不到方法参数名字的，因此只有在类没生成变量调试信息时才需要使用**arg-names**属性来指定参数名，如arg-names="param"表示通知实现方法的参数列表的第一个参数名字为“param”。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

```
1. public void sayBefore(String param);
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

```
1. @Override
2. public void sayBefore(String param) {
3. System.out.println("============say " + param);
4. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

```
1. public void beforeAdvice(String param) {
2. System.out.println("===========before advice param:" + param);
3. }
```

最后在chapter6/advice.xml配置文件中进行如下配置：

```
1. <bean id="helloWorldService"
   class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>
2. <bean id="aspect" class="cn.javass.spring.chapter6.aop.HelloWorldAspect"/>
3. <aop:config>
4. <aop:aspect ref="aspect">
5. <aop:before pointcut="execution( cn.javass...sayBefore(..)) and args(param)"
6. method="beforeAdvice(java.lang.String)"
```

```
7. arg-names="param"/>
8. </aop:aspect>
9. </aop:config>
```

测试代码cn.javass.spring.chapter6.AopTest:

```
1. @Test
2. public void testSchemaBeforeAdvice(){
3. System.out.println("======================================");
4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5. IHelloWorldService helloworldService = ctx.getBean("helloWorldService",
   IHelloWorldService.class);
6. helloworldService.sayBefore("before");
7. System.out.println("======================================");
8. }
```

将输入：

```
==========================================
===========before advice param:before
============say before
==========================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(*cn.javass..*.sayBefore(..)) ”匹配目标方法sayBefore，且使用“args(param)”匹配目标方法只有一个参数且传入的参数类型为通知实现方法中同名的参数类型；

2）目标方法定义：使用method=" beforeAdvice(java.lang.String) "指定前置通知实现方法，且该通知有一个参数类型为java.lang.String参数；

3）目标方法参数命名：其中使用arg-names=" param "指定通知实现方法参数名为“param”，切入点中使用“args(param)”匹配的目标方法参数将自动传递给通知实现方法同名参数。

二、后置返回通知：在切入点选择的方法正常返回时执行，通过<aop:aspect>标签下的<aop:after-returning>标签声明：

```
1. <aop:after-returning pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. method="后置返回通知实现方法名"
3. arg-names="后置返回通知实现方法参数列表参数名字"
4. returning="返回值对应的后置返回通知实现方法参数名"
5. />
```

pointcut和**pointcut-ref**：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

returning：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法执行正常返回后，将把目标方法返回值传给通知方法；returning限定了只有目标方法返回值匹配与通知方法相应参数类型时才能执行后置返回通知，否则不执行，对于returning对应的通知方法参数为Object类型将匹配任何目标返回值。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

```
1. public boolean sayAfterReturning();
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

```
1. @Override
2. public boolean sayAfterReturning() {
3. System.out.println("============after returning");
4. return true;
5. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

```
1. public void afterReturningAdvice(Object retVal) {
2. System.out.println("===========after returning advice retVal:" + retVal);
3. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

```
1. <aop:after-returning pointcut="execution( cn.javass...sayAfterReturning(..))"
2. method="afterReturningAdvice"
3. arg-names="retVal"
4. returning="retVal"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

```
1. @Test
2. public void testSchemaAfterReturningAdvice() {
3. System.out.println("======================================");
4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5. IHelloWorldService helloworldService = ctx.getBean("helloWorldService",
   IHelloWorldService.class);
6. helloworldService.sayAfterReturning();
7. System.out.println("======================================");
```

8. }

将输入：

```
=====================================
============after returning
===========after returning advice retVal:true
=====================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(*cn.javass..*.sayAfterReturning(..)) ”匹配目标方法sayAfterReturning，该方法返回true；

2）目标方法定义：使用method="afterReturningAdvice"指定后置返回通知实现方法；

3）目标方法参数命名：其中使用arg-names="retVal"指定通知实现方法参数名为“retVal”；

4）返回值命名：returning="retVal"用于将目标返回值赋值给通知实现方法参数名为“retVal”的参数上。

三、后置异常通知：在切入点选择的方法抛出异常时执行，通过<aop:aspect>标签下的<aop:after-throwing>标签声明：

1. <aop:after-throwing pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. method="后置异常通知实现方法名"
3. arg-names="后置异常通知实现方法参数列表参数名字"
4. throwing="将抛出的异常赋值给的通知实现方法参数名"/>

pointcut和**pointcut-ref**：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

throwing：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法抛出异常返回后，将把目标方法抛出的异常传给通知方法；throwing限定了只有目标方法抛出的异常匹配与通知方法相应参数异常类型时才能执行后置异常通知，否则不执行，对于throwing对应的通知方法参数为Throwable类型将匹配任何异常。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

1. public void sayAfterThrowing();

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

1. @Override

2. public void sayAfterThrowing() {
3. System.out.println("============before throwing");
4. throw new RuntimeException();
5. }

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

1. public void afterThrowingAdvice(Exception exception) {
2. System.out.println("===========after throwing advice exception:" + exception);
3. }

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

1. <aop:after-throwing pointcut="execution(*cn.javass..*.sayAfterThrowing(..))"
2. method="afterThrowingAdvice"
3. arg-names="exception"
4. throwing="exception"/>

测试代码cn.javass.spring.chapter6.AopTest:

1. @Test(expected = RuntimeException.class)
2. public void testSchemaAfterThrowingAdvice() {
3. System.out.println("======================================");
4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5. IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldService.class);
6. helloworldService.sayAfterThrowing();
7. System.out.println("======================================");
8. }

将输入：

```
======================================
============before throwing
===========after throwing advice exception:java.lang.RuntimeException
======================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(*cn.javass..*.sayAfterThrowing(..))”匹配目标方法sayAfterThrowing，该方法将抛出RuntimeException异常；

2）目标方法定义：使用method="afterThrowingAdvice"指定后置异常通知实现方法；

3）目标方法参数命名：其中使用arg-names="exception"指定通知实现方法参数名为“exception”；

4）异常命名：returning="exception"用于将目标方法抛出的异常赋值给通知实现方法参数名为“exception”的参数上。

四、后置最终通知：在切入点选择的方法返回时执行，不管是正常返回还是抛出异常都执行，通过<aop:aspect>标签下的<aop:after >标签声明：

```
1. <aop:after pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. method="后置最终通知实现方法名"
3. arg-names="后置最终通知实现方法参数列表参数名字"/>
```

pointcut和**pointcut-ref**：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

```
1. public boolean sayAfterFinally();
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

```
1. @Override
2. public boolean sayAfterFinally() {
3. System.out.println("============before finally");
4. throw new RuntimeException();
5. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

```
1. public void afterFinallyAdvice() {
2. System.out.println("===========after finally advice");
3. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

```
1. <aop:after pointcut="execution( cn.javass...sayAfterFinally(..))"
2. method="afterFinallyAdvice"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

```
1. @Test(expected = RuntimeException.class)
2. public void testSchemaAfterFinallyAdvice() {
3. System.out.println("======================================");
4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
```

5. IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldService.class);
6. helloworldService.sayAfterFinally();
7. System.out.println("======================================");
8. }

将输入：

```
======================================
============before finally
===========after finally advice
======================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(*cn.javass*..sayAfterFinally(..))”匹配目标方法sayAfterFinally，该方法将抛出RuntimeException异常；

2）目标方法定义：使用method=" afterFinallyAdvice "指定后置最终通知实现方法。

五、环绕通知：环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知非常强大，可以决定目标方法是否执行，什么时候执行，执行时是否需要替换方法参数，执行完毕是否需要替换返回值，可通过<aop:aspect>标签下的<aop:around >标签声明：

1. <aop:around pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. method="后置最终通知实现方法名"
3. arg-names="后置最终通知实现方法参数列表参数名字"/>

pointcut和**pointcut-ref**：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

环绕通知第一个参数必须是org.aspectj.lang.ProceedingJoinPoint类型，在通知实现方法内部使用ProceedingJoinPoint的proceed()方法使目标方法执行，proceed 方法可以传入可选的Object[]数组，该数组的值将被作为目标方法执行时的参数。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

1. public void sayAround(String param);

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

1. @Override
2. public void sayAround(String param) {

```
3. System.out.println("============around param:" + param);
4. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

```
1. public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
2. System.out.println("===========around before advice");
3. Object retVal = pjp.proceed(new Object[] {"replace"});
4. System.out.println("===========around after advice");
5. return retVal;
6. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

```
1. <aop:around pointcut="execution( cn.javass...sayAround(..))"
2. method="aroundAdvice"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

```
1. @Test
2. public void testSchemaAroundAdvice() {
3. System.out.println("======================================");
4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5. IHelloWorldService helloworldService =
6. ctx.getBean("helloWorldService", IHelloWorldService.class);
7. helloworldService.sayAround("haha");
8. System.out.println("======================================");
9. }
```

将输入：

```
======================================
===========around before advice
============around param:replace
===========around after advice
======================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(*cn.javass..*.sayAround(..))”匹配目标方法sayAround；

2）目标方法定义：使用method="aroundAdvice"指定环绕通知实现方法，在该实现中，第一个方法参数为pjp，类型为ProceedingJoinPoint，其中“Object retVal = pjp.proceed(new Object[] {"replace"});”，用于执行目标方法，且目标方法参数被“new Object[] {"replace"}”替换，最后返回“retVal ”返回值。

3）测试：我们使用“helloworldService.sayAround("haha");”传入参数为“haha”，但最终输出为“replace”，说明参数被替换了。

6.3.4 引入

Spring引入允许为目标对象引入新的接口，通过在< aop:aspect>标签内使用< aop:declare-parents>标签进行引入，定义方式如下：

```
1. <aop:declare-parents
2. types-matching="AspectJ语法类型表达式"
3. implement-interface=引入的接口"
4. default-impl="引入接口的默认实现"
5. delegate-ref="引入接口的默认实现Bean引用"/>
```

types-matching：匹配需要引入接口的目标对象的AspectJ语法类型表达式；

implement-interface：定义需要引入的接口；

default-impl和**delegate-ref**：定义引入接口的默认实现，二者选一，default-impl是接口的默认实现类全限定名，而delegate-ref是默认的实现的委托Bean名；

接下来让我们练习一下吧：

首先定义引入的接口及默认实现：

```
1. package cn.javass.spring.chapter6.service;
2. public interface IIntroductionService {
3. public void induct();
4. }
```

```
1. package cn.javass.spring.chapter6.service.impl;
2. import cn.javass.spring.chapter6.service.IIntroductionService;
3. public class IntroductiondService implements IIntroductionService {
4. @Override
5. public void induct() {
6. System.out.println("=========introduction");
7. }
8. }
```

其次在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

```
1. <aop:declare-parents
2. types-matching="cn.javass..*.IHelloWorldService+"
3. implement-interface="cn.javass.spring.chapter6.service.IIntroductionService"
4. default-impl="cn.javass.spring.chapter6.service.impl.IntroductiondService"/>
```

最后测试一下吧，测试代码cn.javass.spring.chapter6.AopTest：

```
1. @Test
2. public void testSchemaIntroduction() {
3. System.out.println("======================================");
4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5. IIntroductionService introductionService =
6. ctx.getBean("helloWorldService", IIntroductionService.class);
7. introductionService.induct();
8. System.out.println("======================================");
9. }
```

将输入：

```
======================================
=========introduction
======================================
```

分析一下吧：

1）目标对象类型匹配：使用types-matching="cn.javass..*.IHelloWorldService+"匹配IHelloWorldService接口的子类型，如HelloWorldService实现；

2）引入接口定义：通过implement-interface属性表示引入的接口，如“cn.javass.spring.chapter6.service.IIntroductionService”。

3）引入接口的实现：通过default-impl属性指定，如“cn.javass.spring.chapter6.service.impl.IntroductiondService”，也可以使用“delegate-ref”来指定实现的Bean。

4）获取引入接口：如使用“ctx.getBean("helloWorldService", IIntroductionService.class);”可直接获取到引入的接口。

6.3.5 Advisor

Advisor表示只有一个通知和一个切入点的切面，由于Spring AOP都是基于AOP联盟的拦截器模型的环绕通知的，所以引入Advisor来支持各种通知类型（如前置通知等5种），Advisor概念来自于Spring1.2对AOP的支持，在AspectJ中没有相应的概念对应。

Advisor可以使用<aop:config>标签下的<aop:advisor>标签定义：

```
1. <aop:advisor pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. advice-ref="通知API实现引用"/>
```

pointcut和**pointcut-ref**：二者选一，指定切入点表达式；

advice-ref：引用通知API实现Bean，如前置通知接口为MethodBeforeAdvice；

接下来让我们看一下示例吧：

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

```
1. public void sayAdvisorBefore(String param);
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

```
1. @Override
2. public void sayAdvisorBefore(String param) {
3. System.out.println("============say " + param);
4. }
```

第三定义前置通知API实现：

```
1. package cn.javass.spring.chapter6.aop;
2. import java.lang.reflect.Method;
3. import org.springframework.aop.MethodBeforeAdvice;
4. public class BeforeAdviceImpl implements MethodBeforeAdvice {
5. @Override
6. public void before(Method method, Object[] args, Object target) throws Throwable {
7. System.out.println("===========before advice");
8. }
9. }
```

在chapter6/advice.xml配置文件中先添加通知实现Bean定义：

```
1. <bean id="beforeAdvice" class="cn.javass.spring.chapter6.aop.BeforeAdviceImpl"/>
```

然后在<aop:config>标签下，添加Advisor定义，添加时注意顺序：

```
1. <aop:advisor pointcut="execution( cn.javass...sayAdvisorBefore(..))"
2. advice-ref="beforeAdvice"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

```
1. @Test
2. public void testSchemaAdvisor() {
3. System.out.println("======================================");
```

4. ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5. IHelloWorldService helloworldService =
6. ctx.getBean("helloWorldService", IHelloWorldService.class);
7. helloworldService.sayAdvisorBefore("haha");
8. System.out.println("======================================");
9. }

将输入：

```
======================================
===========before advice
============say haha
======================================
```

在此我们只介绍了前置通知API，其他类型的在后边章节介绍。

不推荐使用Advisor，除了在进行事务控制的情况下，其他情况一般不推荐使用该方式，该方式属于侵入式设计，必须实现通知API。

【第六章】 AOP 之 6.4 基于@AspectJ的AOP ——跟我学spring3

Spring除了支持Schema方式配置AOP，还支持注解方式：使用@AspectJ风格的切面声明。

6.4.1 启用对@AspectJ的支持

Spring默认不支持@AspectJ风格的切面声明，为了支持需要使用如下配置：

```
<aop:aspectj-autoproxy/>
```

这样Spring就能发现@AspectJ风格的切面并且将切面应用到目标对象。

6.4.2 声明切面

@AspectJ风格的声明切面非常简单，使用@Aspect注解进行声明：

```
@Aspect()
Public class Aspect{
……
}
```

然后将该切面在配置文件中声明为Bean后，Spring就能自动识别并进行AOP方面的配置：

```
<bean id="aspect" class="……Aspect"/>
```

该切面就是一个POJO，可以在该切面中进行切入点及通知定义，接着往下看吧。

6.4.3 声明切入点

@AspectJ风格的命名切入点使用org.aspectj.lang.annotation包下的@Pointcut+方法（方法必须是返回void类型）实现。

```
@Pointcut(value="切入点表达式", argNames = "参数名列表")
public void pointcutName(……) {}
```

value：指定切入点表达式；

argNames：指定命名切入点方法参数列表参数名字，可以有多个用“，”分隔，这些参数将传递给通知方法同名的参数，同时比如切入点表达式“args(param)”将匹配参数类型为命名切入点方法同名参数指定的参数类型。

pointcutName：切入点名字，可以使用该名字进行引用该切入点表达式。

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)", argNames
public void beforePointcut(String param) {}
```

定义了一个切入点，名字为“beforePointcut”，该切入点将匹配目标方法的第一个参数类型为通知方法实现中参数名为“param”的参数类型。

6.4.4 声明通知

@AspectJ风格的声明通知也支持5种通知类型：

一、前置通知：使用org.aspectj.lang.annotation 包下的@Before注解声明；

```
@Before(value = "切入点表达式或命名切入点", argNames = "参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义。

接下来示例一下吧：

1、定义接口和实现，在此我们就使用Schema风格时的定义；

2、定义切面：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class HelloWorldAspect2 {

}
```

3、定义切入点：

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)", argNames
public void beforePointcut(String param) {}
```

4、定义通知：

```
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
    System.out.println("===========before advice param:" + param);
}
```

5、在chapter6/advice2.xml配置文件中进行如下配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <aop:aspectj-autoproxy/>
  <bean id="helloWorldService"
            class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>

  <bean id="aspect"
             class="cn.javass.spring.chapter6.aop.HelloWorldAspect2"/>

</beans>
```

6、测试代码cn.javass.spring.chapter6.AopTest:

```
@Test
public void testAnnotationBeforeAdvice() {
    System.out.println("======================================");
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice2.xml");
    IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldSe
    helloworldService.sayBefore("before");
    System.out.println("======================================");
}
```

将输出：

```
==========================================

===========before advice param:before

============say before

==========================================
```

切面、切入点、通知全部使用注解完成：

1）使用@Aspect将POJO声明为切面；

2）使用@Pointcut进行命名切入点声明，同时指定目标方法第一个参数类型必须是java.lang.String，对于其他匹配的方法但参数类型不一致的将也是不匹配的，通过argNames = "param"指定了将把该匹配的目标方法参数传递给通知同名的参数上；

3）使用@Before进行前置通知声明，其中value用于定义切入点表达式或引用命名切入点；

4）配置文件需要使用<aop:aspectj-autoproxy/>来开启注解风格的@AspectJ支持；

5）需要将切面注册为Bean，如“aspect”Bean；

6）测试代码完全一样。

二、后置返回通知：使用org.aspectj.lang.annotation 包下的@AfterReturning注解声明；

```
@AfterReturning(
value="切入点表达式或命名切入点",
pointcut="切入点表达式或命名切入点",
argNames="参数列表参数名",
returning="返回值对应参数名")
```

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

returning：与Schema方式配置中的同义。

```
@AfterReturning(
    value="execution(* cn.javass..*.sayBefore(..))",
    pointcut="execution(* cn.javass..*.sayAfterReturning(..))",
    argNames="retVal", returning="retVal")
public void afterReturningAdvice(Object retVal) {
    System.out.println("===========after returning advice retVal:" + retVal);
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterReturningAdvice测试方法。

三、后置异常通知：使用org.aspectj.lang.annotation 包下的@AfterThrowing注解声明；

```
@AfterThrowing (
value="切入点表达式或命名切入点",
pointcut="切入点表达式或命名切入点",
argNames="参数列表参数名",
throwing="异常对应参数名")
```

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

throwing：与Schema方式配置中的同义。

```
@AfterThrowing(
    value="execution(* cn.javass..*.sayAfterThrowing(..))",
    argNames="exception", throwing="exception")
public void afterThrowingAdvice(Exception exception) {
    System.out.println("===========after throwing advice exception:" + exception);
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterThrowingAdvice测试方法。

四、后置最终通知：使用org.aspectj.lang.annotation 包下的@After注解声明；

```
@After (
value="切入点表达式或命名切入点",
argNames="参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

```
@After(value="execution(* cn.javass..*.sayAfterFinally(..))")
public void afterFinallyAdvice() {
    System.out.println("===========after finally advice");
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterFinallyAdvice测试方法。

五、环绕通知：使用org.aspectj.lang.annotation 包下的@Around注解声明；

```
@Around (
value="切入点表达式或命名切入点",
argNames="参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

```
@Around(value="execution(* cn.javass..*.sayAround(..))")
public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("===========around before advice");
    Object retVal = pjp.proceed(new Object[] {"replace"});
    System.out.println("===========around after advice");
    return retVal;
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的annotationAroundAdviceTest测试方法。

6.4.5 引入

@AspectJ风格的引入声明在切面中使用org.aspectj.lang.annotation包下的@DeclareParents声明：

```
@DeclareParents(
value=" AspectJ语法类型表达式",
defaultImpl=引入接口的默认实现类)
private Interface interface;
```

value：匹配需要引入接口的目标对象的AspectJ语法类型表达式；与Schema方式中的types-matching属性同义；

private Interface interface：指定需要引入的接口；

defaultImpl：指定引入接口的默认实现类，没有与Schema方式中的delegate-ref属性同义的定义方式；

```
@DeclareParents(
    value="cn.javass..*.IHelloWorldService+", defaultImpl=cn.javass.spring.chapter6.servi
private IIntroductionService introductionService;
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationIntroduction测试方法。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2471.html】

【第六章】 AOP 之 6.5 AspectJ切入点语法详解 ——跟我学spring3

6.5.1 Spring AOP支持的AspectJ切入点指示符

切入点指示符用来指示切入点表达式目的，，在Spring AOP中目前只有执行方法这一个连接点，Spring AOP支持的AspectJ切入点指示符如下：

execution：用于匹配方法执行的连接点；

within：用于匹配指定类型内的方法执行；

this：用于匹配当前***AOP***代理对象类型的执行方法；注意是***AOP***代理对象的类型匹配，这样就可能包括引入接口也类型匹配；

target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；

args：用于匹配当前执行的方法传入的参数为指定类型的执行方法；

@within：用于匹配所以持有指定注解类型内的方法__；

@target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；

@args：用于匹配当前执行的方法传入的参数持有指定注解的执行；

@annotation：用于匹配当前执行方法持有指定注解的方法；

bean：*Spring AOP*扩展的，*AspectJ*没有对于指示符，用于匹配特定名称的*Bean*__对象的执行方法；

reference pointcut：表示引用其他命名切入点，只有***@ApectJ***风格支持，***Schema***风格不支持。

AspectJ切入点支持的切入点指示符还有： call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this、@withincode；但Spring AOP目前不支持这些指示符，使用这些指示符将抛出IllegalArgumentException异常。这些指示符Spring AOP可能会在以后进行扩展。

6.5.1 命名及匿名切入点

命名切入点可以被其他切入点引用，而匿名切入点是不可以的。

只有@AspectJ支持命名切入点，而Schema风格不支持命名切入点。

如下所示，@AspectJ使用如下方式引用命名切入点：

```
@Pointcut(
    value="execution(* cn.javass..*.sayBefore(java.lang.String)) && args(param)",
    argNames = "param")
public void beforePointcut(String param) {}
                          ↓ 引用命名切入点
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
      System.out.println("===========before advice param:" + param);
}
```

6.5.2 ；类型匹配语法

首先让我们来了解下AspectJ类型匹配的通配符：

- ：匹配任何数量字符；

.. ：（两个点）匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+ ：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

> **java.lang.String** 匹配String类型；
>
> **java.*.String** 匹配java包下的任何"一级子包"下的String类型；
>
> ```
> 如匹配java.lang.String，但不匹配java.lang.ss.String
> ```
>
> **java..*** 匹配java包及任何子包下的任何类型;
>
> ```
> 如匹配java.lang.String、java.lang.annotation.Annotation
> ```
>
> **java.lang.*ing** 匹配任何java.lang包下的以ing结尾的类型；
>
> **java.lang.Number+** 匹配java.lang包下的任何Number的自类型；
>
> ```
> 如匹配java.lang.Integer，也匹配java.math.BigInteger
> ```

接下来再看一下具体的匹配表达式类型吧：

匹配类型：使用如下方式匹配

```
注解？ 类的全限定名字
```

- 注解：可选，类型上持有的注解，如@Deprecated；
- 类的全限定名：必填，可以是任何类全限定名。

匹配方法执行：使用如下方式匹配：

```
注解？ 修饰符? 返回值类型 类型声明?方法名(参数列表) 异常列表？
```

- 注解：可选，方法上持有的注解，如@Deprecated；
- 修饰符：可选，如public、protected；
- 返回值类型：必填，可以是任何类型模式；"*"表示所有类型；
- 类型声明：可选，可以是任何类型模式；
- 方法名：必填，可以使用"*"进行模式匹配；
- 参数列表："()"表示方法没有任何参数；"(..)"表示匹配接受任意个参数的方法，"(..,java.lang.String)"表示匹配接受java.lang.String类型的参数结束，且其前边可以接受有任意个参数的方法；"(java.lang.String,..)" 表示匹配接受java.lang.String类型的参数开始，且其后边可以接受任意个参数的方法；"(*,java.lang.String)" 表示匹配接受java.lang.String类型的参数结束，且其前边接受有一个任意类型参数的方法；
- 异常列表：可选，以"throws 异常全限定名列表"声明，异常全限定名列表如有多个以"，"分割，如throws java.lang.IllegalArgumentException, java.lang.ArrayIndexOutOfBoundsException。

匹配**Bean**名称：可以使用Bean的id或name进行匹配，并且可使用通配符"*"；

6.5.3 组合切入点表达式

AspectJ使用 且（&&）、或（||）、非（！）来组合切入点表达式。

在Schema风格下，由于在XML中使用"&&"需要使用转义字符"&&"来代替之，所以很不方便，因此Spring ASP 提供了and、or、not来代替&&、||、！。

6.5.4 切入点使用示例

一、**execution**：使用"execution(方法表达式)"匹配方法执行；

模式	描述
public (..)	任何公共方法的执行
cn.javass..IPointcutService.()	cn.javass包及所有子包下IPointcutService接口无参方法
cn.javass...(..)	cn.javass包及所有子包下任何类的任何方法

cn.javass..IPointcutService.(*)	cn.javass包及所有子包下IPointcutService接口有一个参数方法
(!cn.javass..IPointcutService+).(..)	非"cn.javass包及所有子包下IPointcutService接型"的任何方法
cn.javass..IPointcutService+.()	cn.javass包及所有子包下IPointcutService接口的的任何无参方法
cn.javass..IPointcut.test*(java.util.Date)	cn.javass包及所有子包下IPointcut前缀类型的 头的只有一个参数类型为java.util.Date的方法， 配是根据方法签名的参数类型进行匹配的，而不 执行时传入的参数类型决定的如定义方法：pub test(Object obj);即使执行时传入java.util.Date， 配的；
cn.javass..IPointcut.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException	cn.javass包及所有子包下IPointcut前缀类型的 法，且抛出IllegalArgumentException和ArrayIndexOutOfBoundsException异常
(cn.javass..IPointcutService+&& java.io.Serializable+).(..)	任何实现了cn.javass包及所有子包下IPointcutS 口和java.io.Serializable接口的类型的任何方法
@java.lang.Deprecated (..)	任何持有@java.lang.Deprecated注解的方法
@java.lang.Deprecated @cn.javass..Secure (..)	任何持有@java.lang.Deprecated和@cn.javass 注解的方法
@(java.lang.Deprecated || cn.javass..Secure) (..)	任何持有@java.lang.Deprecated或@ cn.javas 注解的方法
(@cn.javass..Secure *) (..)	任何返回值类型持有@cn.javass..Secure的方法
*(@cn.javass..Secure *).*(..)	任何定义方法的类型持有@cn.javass..Secure的
(@cn.javass..Secure (*) , @cn.javass..Secure (*))	任何签名带有两个参数的方法，且这个两个参数 Secure标记了，如public void test(@Secure St @Secure String str1);
((@ cn.javass..Secure *))或 (@ cn.javass..Secure *)	任何带有一个参数的方法，且该参数类型持有@ cn.javass..Secure；如public void test(Model m Model类上持有@Secure注解
(@cn.javass..Secure (@cn.javass..Secure *) ,@ cn.javass..Secure (@cn.javass..Secure *))	任何带有两个参数的方法，且这两个参数都被@ cn.javass..Secure标记了；且这两个参数的类型 @ cn.javass..Secure；
(java.util.Map<cn.javass..Model, cn.javass..Model>, ..)	任何带有一个java.util.Map参数的方法，且该参 以< cn.javass..Model, cn.javass..Model >为泛 注意只匹配第一个参数为java.util.Map,不包括子 public void test(HashMap<Model, Model> map str);将不匹配，必须使用" (java.util.HashMap<cn.javass..Model,cn.javass ..)"进行匹配；而public void test(Map map, int i 匹配，因为泛型参数不匹配
	任何带有一个参数（类型为java.util.Collection）

(java.util.Collection<@cn.javass..Secure *>)	法，且该参数类型是有一个泛型参数，该泛型参 上持有@cn.javass..Secure注解；如public void test(Collection<Model> collection);Model类型上 @cn.javass..Secure
(java.util.Set<? extends HashMap>)	任何带有一个参数的方法，且传入的参数类型是 泛型参数，该泛型参数类型继承与HashMap；S **AOP**目前测试不能正常工作
(java.util.List<? super HashMap>)	任何带有一个参数的方法，且传入的参数类型是 泛型参数，该泛型参数类型是HashMap的基类 public voi test(Map map)；**Spring AOP**目前测 常工作
(<@*cn.javass..Secure* >)	任何带有一个参数的方法，且该参数类型是有一 参数，该泛型参数类型上持有@cn.javass..Sec **Spring AOP**目前测试不能正常工作

二、**within**：使用**"within(**类型表达式**)"**匹配指定类型内的方法执行；

模式	描述
within(cn.javass..*)	cn.javass包及子包下的任何方法执行
within(cn.javass..IPointcutService+)	cn.javass包或所有子包下IPointcutService类型及子类型的任何方法
within(@cn.javass..Secure *)	持有cn.javass..Secure注解的任何类型的任何方法必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

三、***this***：使用*"this(*类型全限定名*)"*匹配当前*AOP*代理对象类型的执行方法；注意是*AOP*代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意*this*中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
this(cn.javass.spring.chapter6.service.IPointcutService)	当前AOP对象实现了IPointcutService接口的任何方法
this(cn.javass.spring.chapter6.service.IIntroductionService)	当前AOP对象实现了IIntroductionService接口的任何方法也可能是引入接口

四、***target***：使用*"target(*类型全限定名*)"*匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意*target*中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
target(cn.javass.spring.chapter6.service.IPointcutService)	当前目标对象（非AOP对象）实现了IPointcutService接口的任何方法
target(cn.javass.spring.chapter6.service.IIntroductionService)	当前目标对象（非AOP对象） 实现了IIntroductionService 接口的任何方法不可能是引入接口

五、***args***：使用***"args(参数类型列表)"***匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；*args*__属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

模式	描述
args (java.io.Serializable,..)	任何一个以接受"传入参数类型为 java.io.Serializable" 开头，且其后可跟任意个任意类型的参数的方法执行，args指定的参数类型是在运行时动态匹配的

六、***@within***：使用***"@within(注解类型)__"***匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

模式	描述
@within cn.javass.spring.chapter6.Secure)	任何目标对象对应的类型持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

七、***@target***：使用***"@target(注解类型)__"***匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

模式	描述
@target (cn.javass.spring.chapter6.Secure)	任何目标对象持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

八、***@args***：使用***"@args(注解列表)__"***匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

模式	描述
@args (cn.javass.spring.chapter6.Secure)	任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解cn.javass.spring.chapter6.Secure；动态切入点，类似于arg指示符；

九、**@annotation**：使用**“*@annotation(*注解类型*)__*”**匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

模式	描述
@annotation(cn.javass.spring.chapter6.Secure)	当前执行方法上持有注解 cn.javass.spring.chapter6.Secure将被匹配

十、**bean**：使用**“*bean(Bean id*或名字通配符*)*”**匹配特定名称的**Bean**对象的执行方法；***Spring ASP***扩展的，在***AspectJ***中无相应概念；

模式	描述
bean(*Service)	匹配所有以Service命名（id或name）结尾的Bean

十一、**reference pointcut**：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持，如下所示：

```
@Pointcut(value="bean(*Service)")
private void pointcut1(){}                                        //命名切入点1
@Pointcut(value="@args(cn.javass.spring.chapter6.Secure)")        //命名切入点2
private void pointcut2(){}

@Before(value = "pointcut1() && pointcut2()")                     //引用命名切入点
public void referencePointcutTest1(JoinPoint jp) {
    dump("pointcut1() && pointcut2()", jp);
}
```

比如我们定义如下切面：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class ReferencePointcutAspect {
    @Pointcut(value="execution(* *())")
    public void pointcut() {}
}
```

可以通过如下方式引用：

```
@Before(value = "cn.javass.spring.chapter6.aop.ReferencePointcutAspect.pointcut()")
public void referencePointcutTest2(JoinPoint jp) {}
```

除了可以在@AspectJ风格的切面内引用外，也可以在Schema风格的切面定义内引用，引用方式与@AspectJ完全一样。

到此我们切入点表达式语法示例就介绍完了，我们这些示例几乎包含了日常开发中的所有情况，但当然还有更复杂的语法等等，如果以上介绍的不能满足您的需要，请参考AspectJ文档。

由于测试代码相当长，所以为了节约篇幅本示例代码在cn.javass.spring.chapter6.PointcutTest文件中，需要时请参考该文件。

原创内容，转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2472.html】

【第六章】 AOP 之 6.6 通知参数 ——跟我学 spring3

前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

- 使用**JoinPoint**获取：Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

1) JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

```
package org.aspectj.lang;
import org.aspectj.lang.reflect.SourceLocation;
public interface JoinPoint {
    String toString();         //连接点所在位置的相关信息
    String toShortString();     //连接点所在位置的简短相关信息
    String toLongString();     //连接点所在位置的全部相关信息
    Object getThis();         //返回AOP代理对象
    Object getTarget();       //返回目标对象
    Object[] getArgs();       //返回被通知方法参数列表
    Signature getSignature();  //返回当前连接点签名
    SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置
    String getKind();        //连接点类型
    StaticPart getStaticPart(); //返回连接点静态部分
}
```

2）ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

```
public interface ProceedingJoinPoint extends JoinPoint {
    public Object proceed() throws Throwable;
    public Object proceed(Object[] args) throws Throwable;
}
```

3) JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

```
public interface StaticPart {
    Signature getSignature();    //返回当前连接点签名
    String getKind();          //连接点类型
    int getId();               //唯一标识
    String toString();         //连接点所在位置的相关信息
    String toShortString();     //连接点所在位置的简短相关信息
    String toLongString();     //连接点所在位置的全部相关信息
}
```

使用如下方式在通知方法上声明，必须是在第一个参数，然后使用jp.getArgs()就能获取到被通知方法参数：

```
@Before(value="execution(* sayBefore(*))")
public void before(JoinPoint jp) {}

@Before(value="execution(* sayBefore(*))")
public void before(JoinPoint.StaticPart jp) {}
```

- 自动获取：通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

```
@Before(value="execution(* test(*)) && args(param)", argNames="param")
public void before1(String param) {
    System.out.println("===param:" + param);
}
```

切入点表达式execution(*test()*) && args(param) ：

1）首先execution(*test()*)匹配任何方法名为test，且有一个任何类型的参数；

2）args(param)将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即java.lang.String；如果匹配将把该被通知参数传递给通知方法上同名参数。

其他指示符（除了execution和bean指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【3.1.2构造器注入】中的参数名注入限制：在**class**文件中没生成变量调试信息是获取不到方法参数名字的。

所以我们可以使用策略来确定参数名：

1、如果我们通过“argNames”属性指定了参数名，那么就是要我们指定的；

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(String param) {
    System.out.println("===param:" + param);
}
```

2、如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

3、如果“**class**文件中含有变量调试信息”将使用这些方法签名中的参数名来确定参数名；

```
@Before(value=" args(param)") //不需要argNames了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

4、如果没有“**class**文件中含有变量调试信息”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

```
@Before(value=" args(param)")
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

5、以上策略失败将抛出IllegalArgumentException。

接下来让我们示例一下组合情况吧：

```
@Before(args(param) && target(bean) && @annotation(secure)",
        argNames="jp,param,bean,secure")
public void before5(JoinPoint jp, String param,
 IPointcutService pointcutService, Secure secure) {
……
}
```

该示例的执行步骤如图6-5所示。

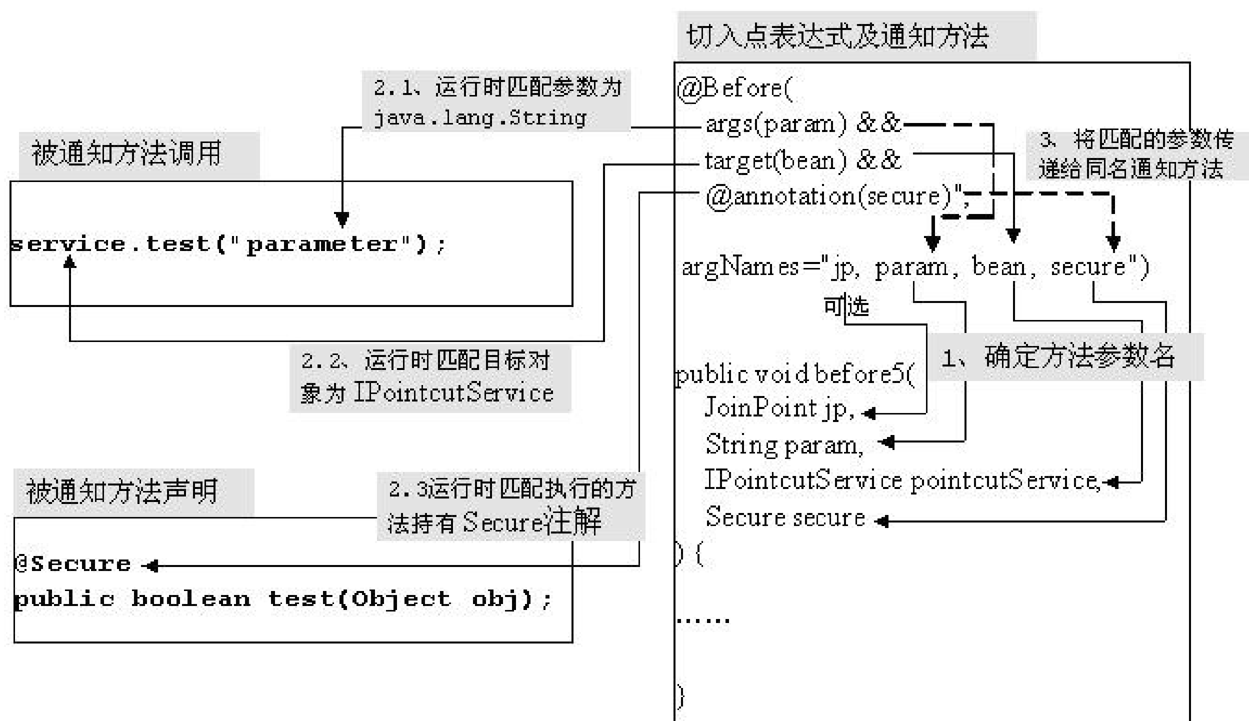


图6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

```
@Pointcut(value="args(param)", argNames="param")
private void pointcut1(String param){}
@Pointcut(value="@annotation(secure)", argNames="secure")
private void pointcut2(Secure secure){}

@Before(value = "pointcut1(param) && pointcut2(secure)",
 argNames="param, secure")
public void before6(JoinPoint jp, String param, Secure secure) {
……
}
```

自此给通知传递参数已经介绍完了，示例代码在cn.javass.spring.chapter6.ParameterTest文件中。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2473.html】

【第六章】 AOP 之 6.7 通知顺序 ——跟我学spring3

如果我们有多个通知想要在同一连接点执行，那执行顺序如何确定呢？Spring AOP使用AspectJ的优先级规则来确定通知执行顺序。总共有两种情况：同一切面中通知执行顺序、不同切面中的通知执行顺序。

首先让我们看下

1） 同一切面中通知执行顺序：如图6-6所示。

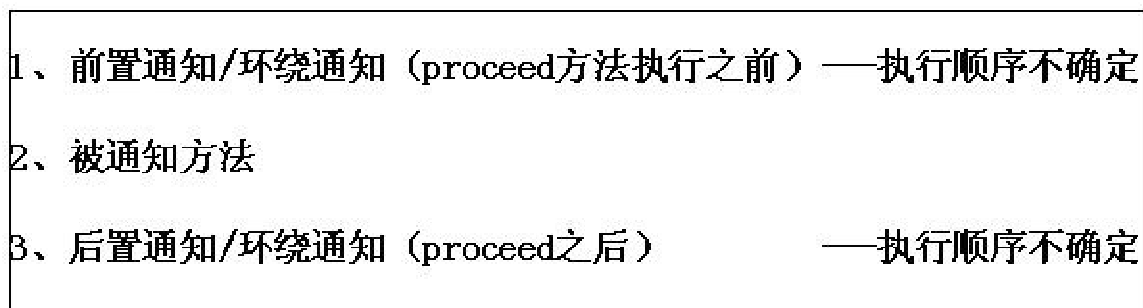


图6-6 同一切面中的通知执行顺序

而如果在同一切面中定义两个相同类型通知（如同是前置通知或环绕通知（proceed之前））并在同一连接点执行时，其执行顺序是未知的，如果确实需要指定执行顺序需要将通知重构到两个切面，然后定义切面的执行顺序。

```
错误“Advice precedence circularity error”：说明AspectJ无法决定通知的执行顺序，只要将通知方法分类并
```

2）不同切面中的通知执行顺序：当定义在不同切面的相同类型的通知需要在同一个连接点执行，如果没指定切面的执行顺序，这两个通知的执行顺序将是未知的。

如果需要他们顺序执行，可以通过指定切面的优先级来控制通知的执行顺序。

Spring中可以通过在切面实现类上实现org.springframework.core.Ordered接口或使用Order注解来指定切面优先级。在多个切面中，Ordered.getValue()方法返回值（或者注解值）较小值的那个切面拥有较高优先级，如图6-7所示。

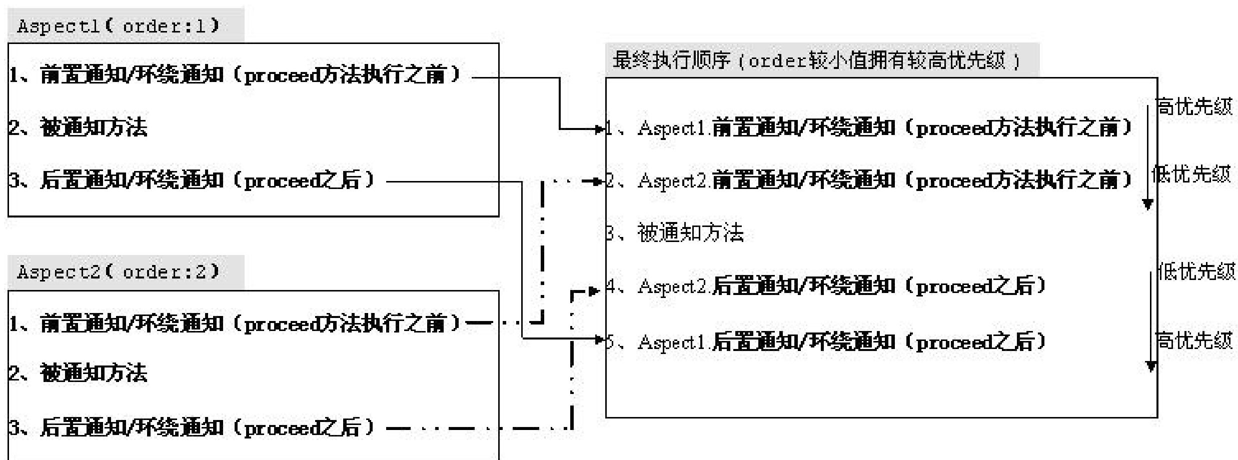
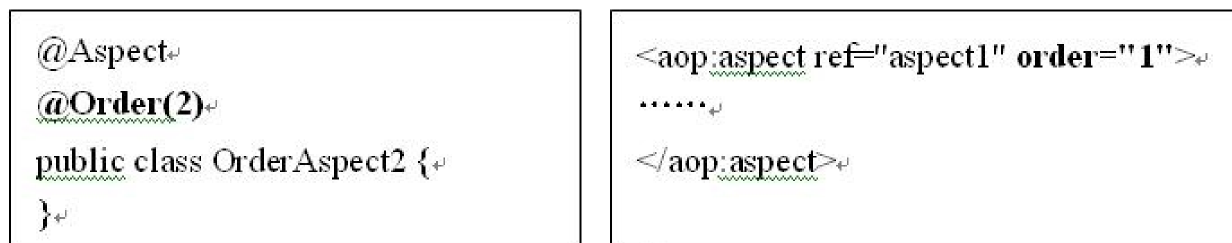


图6-7 两个切面指定了优先级

对于@AspectJ风格和注解风格可分别用以下形式指定优先级：



```
@Aspect
@Order(2)
public class OrderAspect2 {
}
```

```
<aop:aspect ref="aspect1" order="1">
……
</aop:aspect>
```

在此我们不推荐使用实现Ordered接口方法，所以没介绍，示例代码在cn.javass.spring.chapter6. OrderAopTest文件中。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2474.html】

【第六章】 AOP 之 6.8 切面实例化模型 ——跟我学spring3

所谓切面实例化模型指何时实例化切面。

Spring AOP支持AspectJ的singleton、perthis、pertarget实例化模型（目前不支持percflow、percflowbelow 和pertypewithin）。

- **singleton**：即切面只会有一个实例；
- **perthis**：每个切入点表达式匹配的连接点对应的AOP对象都会创建一个新切面实例；
- **pertarget**：每个切入点表达式匹配的连接点对应的目标对象都会创建一个新的切面实例；

默认是singleton实例化模型，Schema风格只支持singleton实例化模型，而@AspectJ风格支持这三种实例化模型。

singleton：使用@Aspect()指定，即默认就是单例实例化模式，在此就不演示示例了。

perthis：每个切入点表达式匹配的连接点对应的AOP对象都会创建一个新的切面实例，使用@Aspect("perthis(切入点表达式)")指定切入点表达式；

如@Aspect("perthis(this(cn.javass.spring.chapter6.service.IIntroductionService))")将对每个匹配“this(cn.javass.spring.chapter6.service.IIntroductionService)”切入点表达式的AOP代理对象创建一个切面实例，注意“IIntroductionService”可能是引入接口。

pertarget：每个切入点表达式匹配的连接点对应的目标对象都会创建一个新的切面实例，使用@Aspect("pertarget(切入点表达式)")指定切入点表达式；

如@Aspect("pertarget(target(cn.javass.spring.chapter6. service.IPointcutService))")将对每个匹配“target(cn.javass.spring.chapter6.service. IPointcutService)”切入点表达式的目标对象创建一个切面，注意“IPointcutService”不可能是引入接口。

在进行切面定义时必须将切面scope定义为“prototype”，如“<bean class="……Aspect" scope="prototype"/>”，否则不能为每个匹配的连接点的目标对象或AOP代理对象创建一个切面。

示例请参考cn.javass.spring.chapter6. InstanceModelTest。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2475.html】

【第六章】 AOP 之 6.9 代理机制 ——跟我学spring3

Spring AOP通过代理模式实现，目前支持两种代理：JDK动态代理、CGLIB代理来创建AOP代理，Spring建议优先使用JDK动态代理。

- **JDK**动态代理：使用java.lang.reflect.Proxy动态代理实现，即提取目标对象的接口，然后对接口创建AOP代理。
- **CGLIB**代理：CGLIB代理不仅能进行接口代理，也能进行类代理，CGLIB代理需要注意以下问题：

不能通知final方法，因为final方法不能被覆盖（CGLIB通过生成子类来创建代理）。

会产生两次构造器调用，第一次是目标类的构造器调用，第二次是CGLIB生成的代理类的构造器调用。如果需要CGLIB代理方法，请确保两次构造器调用不影响应用。

Spring AOP默认首先使用JDK动态代理来代理目标对象，如果目标对象没有实现任何接口将使用CGLIB代理，如果需要强制使用CGLIB代理，请使用如下方式指定：

对于Schema风格配置切面使用如下方式来指定使用CGLIB代理：

```
<aop:config proxy-target-class="true">
</aop:config>
```

而如果使用@AspectJ风格使用如下方式来指定使用CGLIB代理：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3

7.1 概述

7.1.1 JDBC回顾

传统应用程序开发中，进行JDBC编程是相当痛苦的，如下所示：

```
//cn.javass.spring.chapter7\. TraditionalJdbcTest
@Test
public void test() throws Exception {
    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
      conn = getConnection();              //1.获取JDBC连接
                                       //2.声明SQL
      String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
      pstmt = conn.prepareStatement(sql);    //3.预编译SQL
      ResultSet rs = pstmt.executeQuery();   //4.执行SQL
      process(rs);                       //5.处理结果集
      closeResultSet(rs);                 //5.释放结果集
      closeStatement(pstmt);              //6.释放Statement
      conn.commit();                    //8.提交事务
    } catch (Exception e) {
      //9.处理异常并回滚事务
      conn.rollback();
      throw e;
    } finally {
      //10.释放JDBC连接，防止JDBC连接不关闭造成的内存泄漏
      closeConnection(conn);
    }
}
```

以上代码片段具有冗长、重复、容易忘记某一步骤从而导致出错、显示控制事务、显示处理受检查异常等等。

有朋友可能重构出自己的一套JDBC模板，从而能简化日常开发，但自己开发的JDBC模板不够通用，而且对于每一套JDBC模板实现都差不多，从而导致开发人员必须掌握每一套模板。

Spring JDBC提供了一套JDBC抽象框架，用于简化JDBC开发，而且如果各个公司都使用该抽象框架，开发人员首先减少了学习成本，直接上手开发，如图7-1所示。

传统JDBC	Spring JDBC
1.获取JDBC连接	~~1.获取JDBC连接~~
2.声明SQL	**2.声明SQL （√）**
3.预编译SQL	~~3.预编译SQL~~
4.执行SQL	~~4.执行SQL~~
5.处理结果集	**5.处理结果集（√）**
6.释放结果集	~~6.释放结果集~~
7.释放Statement	~~7.释放Statement~~
8.提交事务	~~8.提交事务~~
9.处理异常并回滚事务	~~9.处理异常并回滚事务~~
10.释放JDBC连接	~~10.释放JDBC连接~~
缺点: 1.冗长、重复 2.显示事务控制 3.每个步骤不可获取 4.显示处理受检查异常	优点: 1.简单、简洁 2.Spring事务管理 3.只做需要做的 4.一致的非检查异常体系

图7-1 Spring JDBC与传统JDBC编程对比

7.1.2 Spring对JDBC的支持

Spring通过抽象JDBC访问并提供一致的API来简化JDBC编程的工作量。我们只需要声明**SQL**、调用合适的**Spring JDBC**框架**API**、处理结果集即可。事务由Spring管理，并将JDBC受查异常转换为Spring一致的非受查异常，从而简化开发。

Spring主要提供**JDBC**模板方式、关系数据库对象化方式和**SimpleJdbc**方式三种方式来简化JDBC编程，这三种方式就是Spring JDBC的工作模式：

- **JDBC**模板方式：Spring JDBC框架提供以下几种模板类来简化JDBC编程，实现GoF模板设计模式，将可变部分和非可变部分分离，可变部分采用回调接口方式由用户来实现：如JdbcTemplate、NamedParameterJdbcTemplate、SimpleJdbcTemplate。
- 关系数据库操作对象化方式：Spring JDBC框架提供了将关系数据库操作对象化的表示形式，从而使用户可以采用面向对象编程来完成对数据库的访问；如MappingSqlQuery、SqlUpdate、SqlCall、SqlFunction、StoredProcedure等类。这些类的实现一旦建立即可重用并且是线程安全的。
- **SimpleJdbc**方式：Spring JDBC框架还提供了**SimpleJdbc**方式来简化JDBC编程，SimpleJdbcInsert 、 SimpleJdbcCall用来简化数据库表插入、存储过程或函数访问。

Spring JDBC还提供了一些强大的工具类，如DataSourceUtils来在必要的时候手工获取数据库连接等。

7.1.4 Spring的JDBC架构

Spring JDBC抽象框架由四部分组成：datasource、support、core、object。如图7-2所示。

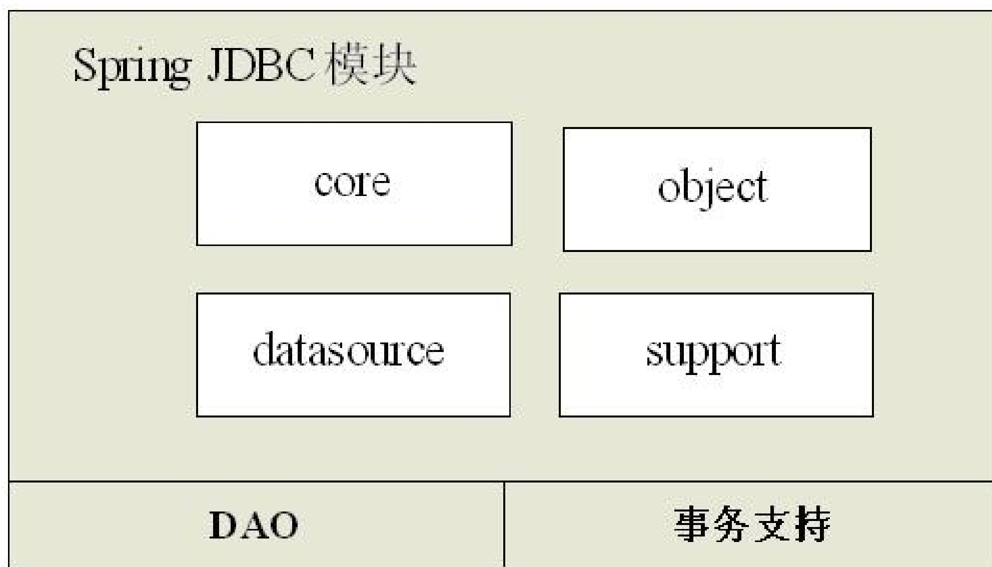


图7-2 Spring JDBC架构图

support包：提供将JDBC异常转换为DAO非检查异常转换类、一些工具类如JdbcUtils等。

datasource包：提供简化访问JDBC 数据源（javax.sql.DataSource实现）工具类，并提供了一些DataSource简单实现类从而能使从这些DataSource获取的连接能自动得到Spring管理事务支持。

core包：提供JDBC模板类实现及可变部分的回调接口，还提供SimpleJdbcInsert等简单辅助类。

object包：提供关系数据库的对象表示形式，如MappingSqlQuery、SqlUpdate、SqlCall、SqlFunction、StoredProcedure等类，该包是基于core包JDBC模板类实现。

原创内容，转载请注明 出处【http://sishuok.com/forum/blogPost/list/0/2489.html】

【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3

7.2 JDBC模板类

7.2.1 概述

Spring JDBC抽象框架core包提供了JDBC模板类，其中JdbcTemplate是core包的核心类，所以其他模板类都是基于它封装完成的，JDBC模板类是第一种工作模式。

JdbcTemplate类通过模板设计模式帮助我们消除了冗长的代码，只做需要做的事情（即可变部分），并且帮我们做哪些固定部分，如连接的创建及关闭。

JdbcTemplate类对可变部分采用回调接口方式实现，如ConnectionCallback通过回调接口返回给用户一个连接，从而可以使用该连接做任何事情、StatementCallback通过回调接口返回给用户一个Statement，从而可以使用该Statement做任何事情等等，还有其他一些回调接口如图7-3所示。

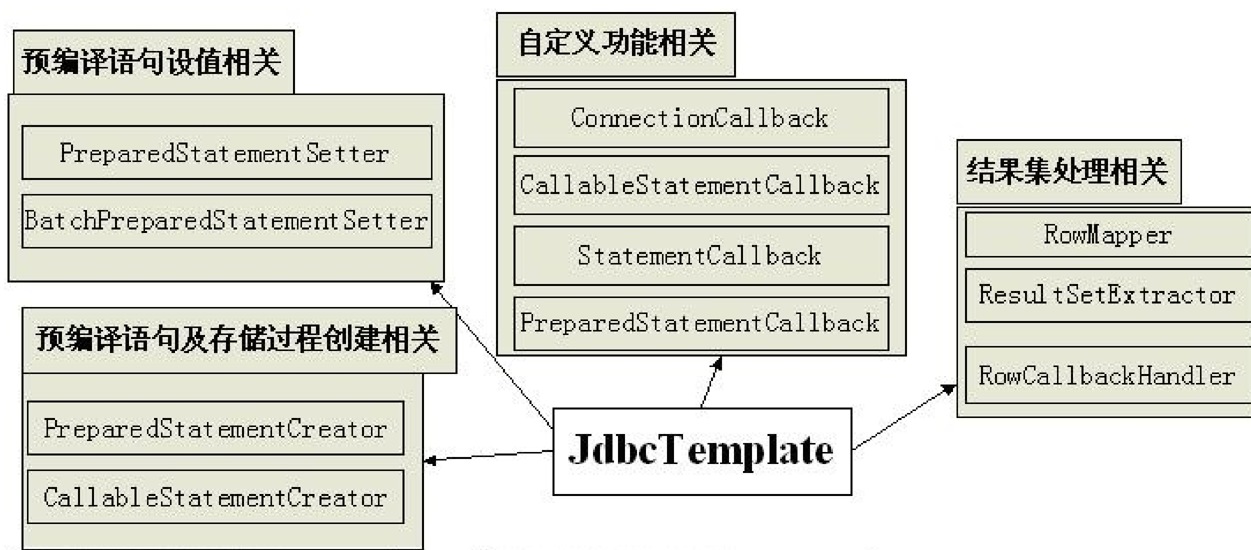


图7-3 JdbcTemplate支持的回调接口

Spring除了提供JdbcTemplate核心类，还提供了基于JdbcTemplate实现的NamedParameterJdbcTemplate类用于支持命名参数绑定、 SimpleJdbcTemplate类用于支持Java5+的可变参数及自动装箱拆箱等特性。

7.2.3 传统JDBC编程替代方案

前边我们已经使用过传统JDBC编程方式，接下来让我们看下Spring JDBC框架提供的更好的解决方案。

1）准备需要的jar包并添加到类路径中：

```
//JDBC抽象框架模块
org.springframework.jdbc-3.0.5.RELEASE.jar
//Spring事务管理及一致的DAO访问及非检查异常模块
org.springframework.transaction-3.0.5.RELEASE.jar
//hsqldb驱动，hsqldb是一个开源的Java实现数据库，请下载hsqldb2.0.0+版本
hsqldb.jar
```

2）传统JDBC编程替代方案：

在使用JdbcTemplate模板类时必须通过DataSource获取数据库连接，Spring JDBC提供了DriverManagerDataSource实现，它通过包装"DriverManager.getConnection"获取数据库连接，具体DataSource相关请参考【7.5.1控制数据库连接】。

```
package cn.javass.spring.chapter7;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
public class JdbcTemplateTest {
    private static JdbcTemplate jdbcTemplate;
    @BeforeClass
    public static void setUpClass() {
        String url = "jdbc:hsqldb:mem:test";
        String username = "sa";
        String password = "";
        DriverManagerDataSource dataSource = new DriverManagerDataSource(url, username, p
        dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    @Test
    public void test() {
        //1.声明SQL
        String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
        jdbcTemplate.query(sql, new RowCallbackHandler() {
            @Override
            public void processRow(ResultSet rs) throws SQLException {
                //2.处理结果集
                String value = rs.getString("TABLE_NAME");
                System.out.println("Column TABLENAME:" + value);
            }
        });
    }
}
```

接下来让我们具体分析一下：

1） **jdbc:hsqldb:mem:test**：表示使用hsqldb内存数据库，数据库名为"test"。

2） **public static void setUpClass()**：使用junit的@BeforeClass注解，表示在所以测试方法之前执行，且只执行一次。在此方法中定义了DataSource并使用DataSource对象创建了JdbcTemplate对象。JdbcTemplate对象是线程安全的。

3） **JdbcTemplate**执行流程：首先定义SQL，其次调用JdbcTemplate方法执行SQL，最后通过RowCallbackHandler回调处理ResultSet结果集。

Spring JDBC解决方法相比传统JDBC编程方式是不是简单多了，是不是只有可变部分需要我们来做，其他的都由Spring JDBC框架来实现了。

接下来让我们深入JdbcTemplate及其扩展吧。

7.2.4 JdbcTemplate

首先让我们来看下如何使用JdbcTemplate来实现增删改查。

一、首先创建表结构：

```
//代码片段(cn.javass.spring.chapter7.JdbcTemplateTest)
@Before
public void setUp() {
   String createTableSql = "create memory table test" + "(id int GENERATED BY DEFAULT AS
   jdbcTemplate.update(createTableSql);
}
@After
public void tearDown() {
  String dropTableSql = "drop table test";
  jdbcTemplate.execute(dropTableSql);
}
```

1） org.junit包下的**@Before**和**@After**分别表示在测试方法之前和之后执行的方法，对于每个测试方法都将执行一次；

2） **create memory table test**表示创建hsqldb内存表，包含两个字段id和name，其中id是具有自增功能的主键，如果有朋友对此不熟悉hsqldb可以换成熟悉的数据库。

二、定义测试骨架，该测试方法将用于实现增删改查测试：

```
@Test
public void testCURD() {
    insert();
    delete();
    update();
    select();
}
```

三、新增测试：

```
private void insert() {
  jdbcTemplate.update("insert into test(name) values('name1')");
  jdbcTemplate.update("insert into test(name) values('name2')");
  Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

四、删除测试：

```
private void delete() {
  jdbcTemplate.update("delete from test where name=?", new Object[]{"name2"});
  Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

五、更新测试：

```
private void update() {
  jdbcTemplate.update("update test set name='name3' where name=?", new Object[]{"name1"})
  Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test where name='
}
```

六、查询测试：

```
private void select() {
  jdbcTemplate.query("select * from test", new RowCallbackHandler(){
    @Override
    public void processRow(ResultSet rs) throws SQLException {
      System.out.print("====id:" + rs.getInt("id"));
      System.out.println(",name:" + rs.getString("name"));
    }
  });
}
```

看完以上示例，大家是否觉得JdbcTemplate简化了我们很多劳动力呢？接下来让我们深入学习一下JdbcTemplate提供的方法。

JdbcTemplate主要提供以下五类方法：

- **execute**方法：可以用于执行任何SQL语句，一般用于执行DDL语句；
- **update**方法及**batchUpdate**方法：update方法用于执行新增、修改、删除等语句；batchUpdate方法用于执行批处理相关语句；
- **query**方法及**queryForXXX**方法：用于执行查询相关语句；
- **call**方法：用于执行存储过程、函数相关语句。

JdbcTemplate类支持的回调类：

- 预编译语句及存储过程创建回调：用于根据JdbcTemplate提供的连接创建相应的语句；

PreparedStatementCreator：通过回调获取JdbcTemplate提供的Connection，由用户使用该Conncetion创建相关的PreparedStatement；

CallableStatementCreator：通过回调获取JdbcTemplate提供的Connection，由用户使用该Conncetion创建相关的CallableStatement；

- 预编译语句设值回调：用于给预编译语句相应参数设值；

PreparedStatementSetter：通过回调获取JdbcTemplate提供的PreparedStatement，由用户来对相应的预编译语句相应参数设值；

BatchPreparedStatementSetter：；类似于PreparedStatementSetter，但用于批处理，需要指定批处理大小；

- 自定义功能回调：提供给用户一个扩展点，用户可以在指定类型的扩展点执行任何数量需要的操作；

ConnectionCallback：通过回调获取JdbcTemplate提供的Connection，用户可在该Connection执行任何数量的操作；

StatementCallback：通过回调获取JdbcTemplate提供的Statement，用户可以在该Statement执行任何数量的操作；

PreparedStatementCallback：通过回调获取JdbcTemplate提供的PreparedStatement，用户可以在该PreparedStatement执行任何数量的操作；

CallableStatementCallback：通过回调获取JdbcTemplate提供的CallableStatement，用户可以在该CallableStatement执行任何数量的操作；

- 结果集处理回调：通过回调处理ResultSet或将ResultSet转换为需要的形式；

RowMapper：用于将结果集每行数据转换为需要的类型，用户需实现方法mapRow(ResultSet rs, int rowNum)来完成将每行数据转换为相应的类型。

RowCallbackHandler：用于处理ResultSet的每一行结果，用户需实现方法processRow(ResultSet rs)来完成处理，在该回调方法中无需执行rs.next()，该操作由JdbcTemplate来执行，用户只需按行获取数据然后处理即可。

ResultSetExtractor：用于结果集数据提取，用户需实现方法extractData(ResultSet rs)来处理结果集，用户必须处理整个结果集；

接下来让我们看下具体示例吧，在示例中不可能介绍到JdbcTemplate全部方法及回调类的使用方法，我们只介绍代表性的，其余的使用都是类似的；

1）预编译语句及存储过程创建回调、自定义功能回调使用：

```
@Test
public void testPpreparedStatement1() {
  int count = jdbcTemplate.execute(new PreparedStatementCreator() {
     @Override
     public PreparedStatement createPreparedStatement(Connection conn)
         throws SQLException {
         return conn.prepareStatement("select count(*) from test");
     }}, new PreparedStatementCallback&lt;Integer&gt;() {
     @Override
     public Integer doInPreparedStatement(PreparedStatement pstmt)
         throws SQLException, DataAccessException {
         pstmt.execute();
         ResultSet rs = pstmt.getResultSet();
         rs.next();
         return rs.getInt(1);
      }});
   Assert.assertEquals(0, count);
}
```

首先使用PreparedStatementCreator创建一个预编译语句，其次由JdbcTemplate通过PreparedStatementCallback回调传回，由用户决定如何执行该PreparedStatement。此处我们使用的是execute方法。

2）预编译语句设值回调使用：

```
@Test
public void testPreparedStatement2() {
  String insertSql = "insert into test(name) values (?)";
  int count = jdbcTemplate.update(insertSql, new PreparedStatementSetter() {
      @Override
      public void setValues(PreparedStatement pstmt) throws SQLException {
          pstmt.setObject(1, "name4");
  }});
  Assert.assertEquals(1, count);
  String deleteSql = "delete from test where name=?";
  count = jdbcTemplate.update(deleteSql, new Object[] {"name4"});
  Assert.assertEquals(1, count);
}
```

通过JdbcTemplate的int update(String sql, PreparedStatementSetter pss)执行预编译sql，其中sql参数为“insert into test(name) values (?) ”，该sql有一个占位符需要在执行前设值，PreparedStatementSetter实现就是为了设值，使用setValues(PreparedStatement pstmt)回调方法设值相应的占位符位置的值。JdbcTemplate也提供一种更简单的方式“update(String sql, Object... args)”来实现设值，所以只要当使用该种方式不满足需求时才应使用PreparedStatementSetter。

3）结果集处理回调：

```
@Test
public void testResultSet1() {
  jdbcTemplate.update("insert into test(name) values('name5')");
  String listSql = "select * from test";
  List result = jdbcTemplate.query(listSql, new RowMapper&lt;Map&gt;() {
      @Override
      public Map mapRow(ResultSet rs, int rowNum) throws SQLException {
          Map row = new HashMap();
          row.put(rs.getInt("id"), rs.getString("name"));
          return row;
  }});
  Assert.assertEquals(1, result.size());
  jdbcTemplate.update("delete from test where name='name5'");
}
```

RowMapper接口提供mapRow(ResultSet rs, int rowNum)方法将结果集的每一行转换为一个Map，当然可以转换为其他类，如表的对象画形式。

```
@Test
public void testResultSet2() {
  jdbcTemplate.update("insert into test(name) values('name5')");
  String listSql = "select * from test";
  final List result = new ArrayList();
  jdbcTemplate.query(listSql, new RowCallbackHandler() {
      @Override
      public void processRow(ResultSet rs) throws SQLException {
          Map row = new HashMap();
          row.put(rs.getInt("id"), rs.getString("name"));
          result.add(row);
  }});
  Assert.assertEquals(1, result.size());
  jdbcTemplate.update("delete from test where name='name5'");
}
```

RowCallbackHandler接口也提供方法processRow(ResultSet rs)，能将结果集的行转换为需要的形式。

```
@Test
public void testResultSet3() {
  jdbcTemplate.update("insert into test(name) values('name5')");
  String listSql = "select * from test";
  List result = jdbcTemplate.query(listSql, new ResultSetExtractor&lt;List&gt;() {
      @Override
      public List extractData(ResultSet rs)
     throws SQLException, DataAccessException {
          List result = new ArrayList();
          while(rs.next()) {
              Map row = new HashMap();
              row.put(rs.getInt("id"), rs.getString("name"));
              result.add(row);
           }
           return result;
  }});
  Assert.assertEquals(0, result.size());
  jdbcTemplate.update("delete from test where name='name5'");
}
```

ResultSetExtractor使用回调方法extractData(ResultSet rs)提供给用户整个结果集，让用户决定如何处理该结果集。

当然JdbcTemplate提供更简单的queryForXXX方法，来简化开发：

```
//1.查询一行数据并返回int型结果
jdbcTemplate.queryForInt("select count(*) from test");
//2\. 查询一行数据并将该行数据转换为Map返回
jdbcTemplate.queryForMap("select * from test where name='name5'");
//3.查询一行任何类型的数据，最后一个参数指定返回结果类型
jdbcTemplate.queryForObject("select count(*) from test", Integer.class);
//4.查询一批数据，默认将每行数据转换为Map
jdbcTemplate.queryForList("select * from test");
//5.只查询一列数据列表，列类型是String类型，列名字是name
jdbcTemplate.queryForList("
select name from test where name=?", new Object[]{"name5"}, String.class);
//6.查询一批数据，返回为SqlRowSet，类似于ResultSet，但不再绑定到连接上
SqlRowSet rs = jdbcTemplate.queryForRowSet("select * from test");
```

3） 存储过程及函数回调：

首先修改JdbcTemplateTest的setUp方法，修改后如下所示：

```
@Before
public void setUp() {
    String createTableSql = "create memory table test" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
    jdbcTemplate.update(createTableSql);

    String createHsqldbFunctionSql =
      "CREATE FUNCTION FUNCTION_TEST(str CHAR(100)) " +
      "returns INT begin atomic return length(str);end";
    jdbcTemplate.update(createHsqldbFunctionSql);
    String createHsqldbProcedureSql =
      "CREATE PROCEDURE PROCEDURE_TEST" +
      "(INOUT inOutName VARCHAR(100), OUT outId INT) " +
      "MODIFIES SQL DATA " +
      "BEGIN ATOMIC " +
      "  insert into test(name) values (inOutName); " +
      "  SET outId = IDENTITY(); " +
      "  SET inOutName = 'Hello,' + inOutName; " +
    "END";
    jdbcTemplate.execute(createHsqldbProcedureSql);
}
```

其中CREATE FUNCTION FUNCTION_TEST用于创建自定义函数，CREATE PROCEDURE PROCEDURE_TEST用于创建存储过程，注意这些创建语句是数据库相关的，本示例中的语句只适用于HSQLDB数据库。

其次修改JdbcTemplateTest的tearDown方法，修改后如下所示：

```
@After
public void tearDown() {
    jdbcTemplate.execute("DROP FUNCTION FUNCTION_TEST");
    jdbcTemplate.execute("DROP PROCEDURE PROCEDURE_TEST");
    String dropTableSql = "drop table test";
    jdbcTemplate.execute(dropTableSql);
}
```

其中drop语句用于删除创建的存储过程、自定义函数及数据库表。

接下来看一下hsqldb如何调用自定义函数：

```
@Test
public void testCallableStatementCreator1() {
    final String callFunctionSql = "{call FUNCTION_TEST(?)}";
    List&lt;SqlParameter&gt; params = new ArrayList&lt;SqlParameter&gt;();
    params.add(new SqlParameter(Types.VARCHAR));
    params.add(new SqlReturnResultSet("result",
       new ResultSetExtractor&lt;Integer&gt;() {
           @Override
           public Integer extractData(ResultSet rs) throws SQLException,
               DataAccessException {
               while(rs.next()) {
                   return rs.getInt(1);
               }
              return 0;
       }));
    Map&lt;String, Object&gt; outValues = jdbcTemplate.call(
       new CallableStatementCreator() {
            @Override
            public CallableStatement createCallableStatement(Connection conn) throws SQLE
              CallableStatement cstmt = conn.prepareCall(callFunctionSql);
              cstmt.setString(1, "test");
              return cstmt;
    }}, params);
    Assert.assertEquals(4, outValues.get("result"));
}
```

- **{call FUNCTION_TEST(?)}**：定义自定义函数的sql语句，注意hsqldb {?= call …}和{call …}含义是一样的，而比如mysql中两种含义是不一样的；
- params：用于描述自定义函数占位符参数或命名参数类型；SqlParameter用于描述IN类型参数、SqlOutParameter用于描述OUT类型参数、SqlInOutParameter用于描述INOUT类型参数、SqlReturnResultSet用于描述调用存储过程或自定义函数返回的ResultSet类型数据，其中SqlReturnResultSet需要提供结果集处理回调用于将结果集转换为相应的形式，hsqldb自定义函数返回值是ResultSet类型。
- **CallableStatementCreator**：提供Connection对象用于创建CallableStatement对象
- outValues：调用call方法将返回类型为Map<String, Object>对象；
- **outValues.get("result")**：获取结果，即通过SqlReturnResultSet对象转换过的数据；其中SqlOutParameter、SqlInOutParameter、SqlReturnResultSet指定的name用于从call执行后返回的Map中获取相应的结果，即name是Map的键。

注：因为hsqldb {?= call …}和{call …}含义是一样的，因此调用自定义函数将返回一个包含结果的ResultSet。

最后让我们示例下mysql如何调用自定义函数：

```
@Test
public void testCallableStatementCreator2() {
    JdbcTemplate mysqlJdbcTemplate = new JdbcTemplate(getMysqlDataSource);
    //2.创建自定义函数
String createFunctionSql =
    "CREATE FUNCTION FUNCTION_TEST(str VARCHAR(100)) " +
     "returns INT return LENGTH(str)";
String dropFunctionSql = "DROP FUNCTION IF EXISTS FUNCTION_TEST";
mysqlJdbcTemplate.update(dropFunctionSql);
mysqlJdbcTemplate.update(createFunctionSql);
//3.准备sql,mysql支持{?= call …}
final String callFunctionSql = "{?= call FUNCTION_TEST(?)}";
//4.定义参数
List&lt;SqlParameter&gt; params = new ArrayList&lt;SqlParameter&gt;();
params.add(new SqlOutParameter("result", Types.INTEGER));
params.add(new SqlParameter("str", Types.VARCHAR));
Map&lt;String, Object&gt; outValues = mysqlJdbcTemplate.call(
new CallableStatementCreator() {
    @Override
    public CallableStatement createCallableStatement(Connection conn) throws SQLException
      CallableStatement cstmt = conn.prepareCall(callFunctionSql);
      cstmt.registerOutParameter(1, Types.INTEGER);
      cstmt.setString(2, "test");
        return cstmt;
    }}, params);
   Assert.assertEquals(4, outValues.get("result"));
}
public DataSource getMysqlDataSource() {
    String url = "jdbc:mysql://localhost:3306/test";
    DriverManagerDataSource dataSource =
        new DriverManagerDataSource(url, "root", "");     dataSource.setDriverClassName("
    return dataSource;
}
```

- **getMysqlDataSource**：首先启动mysql（本书使用5.4.3版本），其次登录mysql创建test数据库（“create database test;”），在进行测试前，请先下载并添加mysql-connector-java-5.1.10.jar到classpath；
- **{?= call FUNCTION_TEST(?)}**：可以使用{?= call …}形式调用自定义函数；
- **params**：无需使用SqlReturnResultSet提取结果集数据，而是使用SqlOutParameter来描述自定义函数返回值；
- **CallableStatementCreator**：同上个例子含义一样；
- **cstmt.registerOutParameter(1, Types.INTEGER)**：将OUT类型参数注册为JDBC类型Types.INTEGER，此处即返回值类型为Types.INTEGER。
- **outValues.get("result")**：获取结果，直接返回Integer类型，比hsqldb简单多了吧。

最后看一下如何如何调用存储过程：

```
@Test
public void testCallableStatementCreator3() {
    final String callProcedureSql = "{call PROCEDURE_TEST(?, ?)}";
    List&lt;SqlParameter&gt; params = new ArrayList&lt;SqlParameter&gt;();
    params.add(new SqlInOutParameter("inOutName", Types.VARCHAR));
    params.add(new SqlOutParameter("outId", Types.INTEGER));
    Map&lt;String, Object&gt; outValues = jdbcTemplate.call(
      new CallableStatementCreator() {
        @Override
        public CallableStatement createCallableStatement(Connection conn) throws SQLExcep
          CallableStatement cstmt = conn.prepareCall(callProcedureSql);
          cstmt.registerOutParameter(1, Types.VARCHAR);
          cstmt.registerOutParameter(2, Types.INTEGER);
          cstmt.setString(1, "test");
          return cstmt;
    }}, params);
    Assert.assertEquals("Hello,test", outValues.get("inOutName"));
    Assert.assertEquals(0, outValues.get("outId"));
}
```

- **{call PROCEDURE_TEST(?, ?)}**：定义存储过程sql；
- **params**：定义存储过程参数；SqlInOutParameter描述INOUT类型参数、SqlOutParameter描述OUT类型参数；
- CallableStatementCreator：用于创建CallableStatement，并设值及注册OUT参数类型；
- outValues：通过SqlInOutParameter及SqlOutParameter参数定义的name来获取存储过程结果。

JdbcTemplate类还提供了很多便利方法，在此就不一一介绍了，但这些方法是由规律可循的，第一种就是提供回调接口让用户决定做什么，第二种可以认为是便利方法（如queryForXXX），用于那些比较简单的操作。

7.2.4 NamedParameterJdbcTemplate

NamedParameterJdbcTemplate类是基于JdbcTemplate类，并对它进行了封装从而支持命名参数特性。

NamedParameterJdbcTemplate主要提供以下三类方法：execute方法、query及queryForXXX方法、update及batchUpdate方法。

首先让我们看个例子吧：

```
@Test
public void testNamedParameterJdbcTemplate1() {
NamedParameterJdbcTemplate namedParameterJdbcTemplate = null;
//namedParameterJdbcTemplate =
//    new NamedParameterJdbcTemplate(dataSource);
namedParameterJdbcTemplate =
new NamedParameterJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(:name)";
    String selectSql = "select * from test where name=:name";
    String deleteSql = "delete from test where name=:name";
    Map&lt;String, Object&gt; paramMap = new HashMap&lt;String, Object&gt;();
    paramMap.put("name", "name5");
    namedParameterJdbcTemplate.update(insertSql, paramMap);
    final List&lt;Integer&gt; result = new ArrayList&lt;Integer&gt;();
namedParameterJdbcTemplate.query(selectSql, paramMap,
new RowCallbackHandler() {
        @Override
        public void processRow(ResultSet rs) throws SQLException {
            result.add(rs.getInt("id"));
        }
    });
Assert.assertEquals(1, result.size());
SqlParameterSource paramSource = new MapSqlParameterSource(paramMap);
namedParameterJdbcTemplate.update(deleteSql, paramSource);
}
```

接下来让我们分析一下代码吧：

1）**NamedParameterJdbcTemplate**初始化：可以使用DataSource或JdbcTemplate 对象作为构造器参数初始化；

2）**insert into test(name) values(:name)**：其中“:name”就是命名参数；

3） **update(insertSql, paramMap)**：其中paramMap是一个Map类型，包含键为“name”，值为“name5”的键值对，也就是为命名参数设值的数据；

4）**query(selectSql, paramMap, new RowCallbackHandler()……)**：类似于JdbcTemplate中介绍的，唯一不同是需要传入paramMap来为命名参数设值；

5）**update(deleteSql, paramSource)**：类似于“update(insertSql, paramMap)”，但使用SqlParameterSource参数来为命名参数设值，此处使用MapSqlParameterSource实现，其就是简单封装java.util.Map。

NamedParameterJdbcTemplate类为命名参数设值有两种方式：java.util.Map和SqlParameterSource：

1）**java.util.Map**：使用Map键数据来对于命名参数，而Map值数据用于设值；

2）**SqlParameterSource**：可以使用SqlParameterSource实现作为来实现为命名参数设值，默认有MapSqlParameterSource和BeanPropertySqlParameterSource实现；MapSqlParameterSource实现非常简单，只是封装了java.util.Map；而BeanPropertySqlParameterSource封装了一个JavaBean对象，通过JavaBean对象属性来决定命名参数的值。

```
package cn.javass.spring.chapter7;
public class UserModel {
    private int id;
    private String myName;
    //省略getter和setter
}
```

```
@Test
public void testNamedParameterJdbcTemplate2() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate = null;
    namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(jdbcTemplate);
    UserModel model = new UserModel();
    model.setMyName("name5");
    String insertSql = "insert into test(name) values(:myName)";
    SqlParameterSource paramSource = new BeanPropertySqlParameterSource(model);
    namedParameterJdbcTemplate.update(insertSql, paramSource);
}
```

可以看出BeanPropertySqlParameterSource使用能减少很多工作量，但命名参数必须和JavaBean属性名称相对应才可以。

7.2.5 SimpleJdbcTemplate

SimpleJdbcTemplate类也是基于JdbcTemplate类，但利用Java5+的可变参数列表和自动装箱和拆箱从而获取更简洁的代码。

SimpleJdbcTemplate主要提供两类方法：query及queryForXXX方法、update及batchUpdate方法。

首先让我们看个例子吧：

```
//定义UserModel的RowMapper
package cn.javass.spring.chapter7;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
public class UserRowMapper implements RowMapper&lt;UserModel&gt; {
    @Override
    public UserModel mapRow(ResultSet rs, int rowNum) throws SQLException {
        UserModel model = new UserModel();
        model.setId(rs.getInt("id"));
        model.setMyName(rs.getString("name"));
        return model;
    }
}
```

```
@Test
public void testSimpleJdbcTemplate() {
    //还支持DataSource和NamedParameterJdbcTemplate作为构造器参数
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(id, name) values(?, ?)";
    simpleJdbcTemplate.update(insertSql, 10, "name5");
    String selectSql = "select * from test where id=? and name=?";
    List&lt;Map&lt;String, Object&gt;&gt; result = simpleJdbcTemplate.queryForList(select
    Assert.assertEquals(1, result.size());
    RowMapper&lt;UserModel&gt; mapper = new UserRowMapper();
    List&lt;UserModel&gt; result2 = simpleJdbcTemplate.query(selectSql, mapper, 10, "name
    Assert.assertEquals(1, result2.size());
}
```

1）**SimpleJdbcTemplate**初始化：可以使用DataSource、JdbcTemplate或NamedParameterJdbcTemplate对象作为构造器参数初始化；

2）**update(insertSql, 10, "name5")**：采用Java5+可变参数列表从而代替new Object[]{10, "name5"}方式；

3）**query(selectSql, mapper, 10, "name5")**：使用Java5+可变参数列表及RowMapper回调并利用泛型特性来指定返回值类型（List<UserModel>）。

SimpleJdbcTemplate类还支持命名参数特性，如queryForList(String sql, SqlParameterSource args)和queryForList(String sql, Map<String, ?> args) ，类似于NamedParameterJdbcTemplate中使用，在此就不介绍了。

> | > > 注：SimpleJdbcTemplate还提供类似于ParameterizedRowMapper 用于泛型特性的支持，ParameterizedRowMapper是RowMapper的子类，但从Spring 3.0由于允许环境需要Java5+，因此不再需要ParameterizedRowMapper ，而可以直接使用RowMapper； > > **query(String sql, ParameterizedRowMapper<T> rm, SqlParameterSource args)** > > **query(String sql, RowMapper<T> rm, Object... args)** **//**直接使用该语句 > > |

> | > > SimpleJdbcTemplate还提供如下方法用于获取JdbcTemplate和NamedParameterJdbcTemplate： > > 1）获取JdbcTemplate对象方法：JdbcOperations是JdbcTemplate的接口 > > **JdbcOperations getJdbcOperations()** > > 2）获取NamedParameterJdbcTemplate对象方法：NamedParameterJdbcOperations是NamedParameterJdbcTemplate的接口 > > **NamedParameterJdbcOperations getNamedParameterJdbcOperations()** > > |

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2490.html】

【第七章】 对JDBC的支持 之 7.3 关系数据库操作对象化 ——跟我学spring3

7.3.1 概述

所谓关系数据库对象化其实就是用面向对象方式表示关系数据库操作，从而可以复用。

Spring JDBC框架将数据库操作封装为一个RdbmsOperation，该对象是线程安全的、可复用的对象，是所有数据库对象的父类。而SqlOperation继承了RdbmsOperation，代表了数据库SQL操作，如select、update、call等，如图7-4所示。

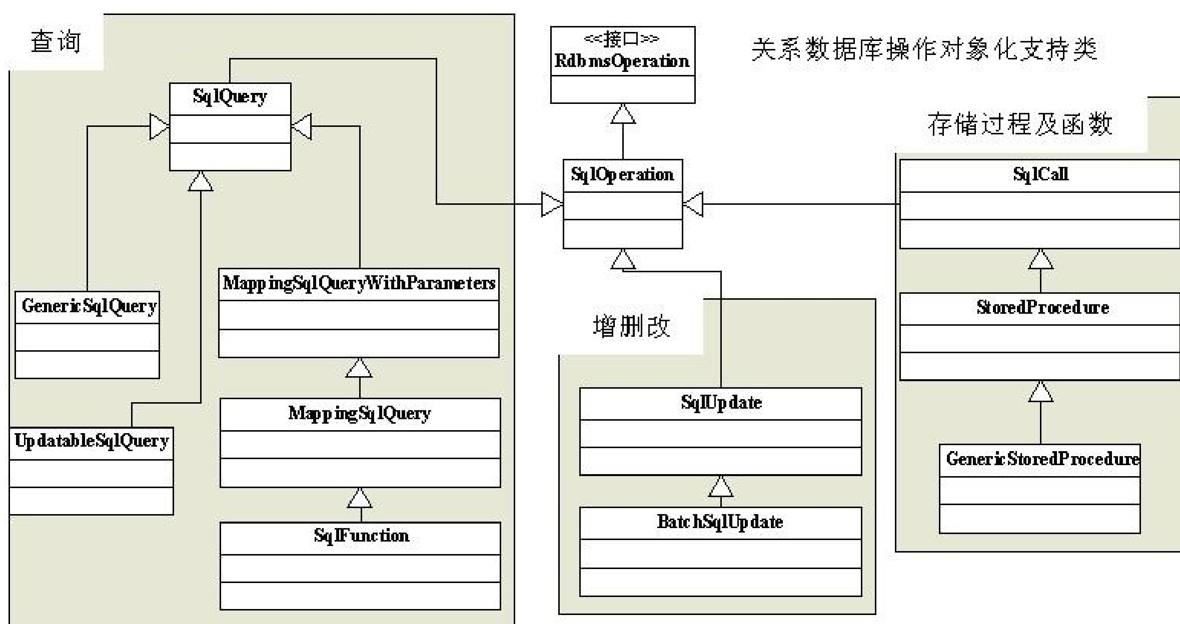


图7-4 关系数据库操作对象化支持类

数据库操作对象化只要有以下几种类型，所以类型是线程安全及可复用的：

- 查询：将数据库操作select封装为对象，查询操作的基类是SqlQuery，所有查询都可以使用该类表示，Spring JDBC还提供了一些更容易使用的MappingSqlQueryWithParameters和MappingSqlQuery用于将结果集映射为Java对象，查询对象类还提供了两个扩展UpdatableSqlQuery和SqlFunction；
- 更新：即增删改操作，将数据库操作insert 、update、delete封装为对象，增删改基类是SqlUpdate，当然还提供了BatchSqlUpdate用于批处理；
- 存储过程及函数：将存储过程及函数调用封装为对象，基类是SqlCall类，提供了StoredProcedure实现。

7.3.2 查询

1）SqlQuery：需要覆盖如下方法来定义一个RowMapper，其中parameters参数表示命名参数或占位符参数值列表，而context是由用户传入的上下文数据。

```
RowMapper<T> newRowMapper(Object[] parameters, Map context)
```

SqlQuery提供两类方法：

- execute及executeByNamedParam方法：用于查询多行数据，其中executeByNamedParam用于支持命名参数绑定参数；
- findObject及findObjectByNamedParam方法：用于查询单行数据，其中findObjectByNamedParam用于支持命名参数绑定。

演示一下SqlQuery如何使用：

```
@Test
public void testSqlQuery() {
    SqlQuery query = new UserModelSqlQuery(jdbcTemplate);
    List<UserModel> result = query.execute("name5");
    Assert.assertEquals(0, result.size());
}
```

从测试代码可以SqlQuery使用非常简单，创建SqlQuery实现对象，然后调用相应的方法即可，接下来看一下SqlQuery实现：

```
package cn.javass.spring.chapter7;
//省略import
public class UserModelSqlQuery extends SqlQuery<UserModel> {
    public UserModelSqlQuery(JdbcTemplate jdbcTemplate) {
        //super.setDataSource(jdbcTemplate.getDataSource());
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("select * from test where name=?");
        super.declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
    @Override
    protected RowMapper<UserModel> newRowMapper(Object[] parameters, Map context) {
        return new UserRowMapper();
    }
}
```

从测试代码可以看出，具体步骤如下：

一、setJdbcTemplate/ setDataSource：首先设置数据源或JdbcTemplate；

二、setSql("select * from test where name=?")：定义sql语句，所以定义的sql语句都将被编译为PreparedStatement；

三、declareParameter(new SqlParameter(Types.VARCHAR))：对PreparedStatement参数描述，使用SqlParameter来描述参数类型，支持命名参数、占位符描述；

对于命名参数可以使用如new SqlParameter("name", Types.VARCHAR)描述；注意占位符参数描述必须按占位符参数列表的顺序进行描述；

四、编译：可选，当执行相应查询方法时会自动编译，用于将sql编译为PreparedStatement，对于编译的SqlQuery不能再对参数进行描述了。

五、以上步骤是不可变的，必须按顺序执行。

2）MappingSqlQuery：用于简化SqlQuery中RowMapper创建，可以直接在实现mapRow(ResultSet rs, int rowNum)来将行数据映射为需要的形式；

MappingSqlQuery所有查询方法完全继承于SqlQuery。

演示一下MappingSqlQuery如何使用：

```
@Test
public void testMappingSqlQuery() {
    jdbcTemplate.update("insert into test(name) values('name5')");
    SqlQuery<UserModel> query = new UserModelMappingSqlQuery(jdbcTemplate);
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    UserModel result = query.findObjectByNamedParam(paramMap);
    Assert.assertNotNull(result);
}
```

MappingSqlQuery使用和SqlQuery完全一样，创建MappingSqlQuery实现对象，然后调用相应的方法即可，接下来看一下MappingSqlQuery实现，findObjectByNamedParam方法用于执行命名参数查询：

```
package cn.javass.spring.chapter7;
//省略import
public class UserModelMappingSqlQuery extends MappingSqlQuery<UserModel> {
  public UserModelMappingSqlQuery(JdbcTemplate jdbcTemplate) {
    super.setDataSource(jdbcTemplate.getDataSource());
    super.setSql("select * from test where name=:name");
    super.declareParameter(new SqlParameter("name", Types.VARCHAR));
    compile();
  }
  @Override
  protected UserModel mapRow(ResultSet rs, int rowNum) throws SQLException {
    UserModel model = new UserModel();
    model.setId(rs.getInt("id"));
    model.setMyName(rs.getString("name"));
    return model;
  }
}
```

和SqlQuery唯一不同的是使用mapRow来讲每行数据转换为需要的形式，其他地方完全一样。

1）**UpdatableSqlQuery**：提供可更新结果集查询支持，子类实现updateRow(ResultSet rs, int rowNum, Map context)对结果集进行更新。

2）**GenericSqlQuery**：提供setRowMapperClass(Class rowMapperClass)方法用于指定RowMapper实现，在此就不演示了。具体请参考testGenericSqlQuery()方法。

3） **SqlFunction**：SQL"函数"包装器，用于支持那些返回单行结果集的查询。该类主要用于返回单行单列结果集。

```
@Test
public void testSqlFunction() {
    jdbcTemplate.update("insert into test(name) values('name5')");
    String countSql = "select count(*) from test";
    SqlFunction<Integer> sqlFunction1 = new SqlFunction<Integer>(jdbcTemplate.getDataSour
    Assert.assertEquals(1, sqlFunction1.run());
    String selectSql = "select name from test where name=?";
    SqlFunction<String> sqlFunction2 = new SqlFunction<String>(jdbcTemplate.getDataSource
    sqlFunction2.declareParameter(new SqlParameter(Types.VARCHAR));
    String name = (String) sqlFunction2.runGeneric(new Object[] {"name5"});
    Assert.assertEquals("name5", name);
}
```

如代码所示，SqlFunction初始化时需要DataSource和相应的sql语句，如果有参数需要使用declareParameter对参数类型进行描述；run方法默认返回int型，当然也可以使用runGeneric返回其他类型，如String等。

7.3.3 更新

SqlUpdate类用于支持数据库更新操作，即增删改（insert、delete、update）操作，该方法类似于SqlQuery，只是职责不一样。

SqlUpdate提供了update及updateByNamedParam方法用于数据库更新操作，其中updateByNamedParam用于命名参数类型更新。

演示一下SqlUpdate如何使用：

```
package cn.javass.spring.chapter7;
//省略import
public class InsertUserModel extends SqlUpdate {
    public InsertUserModel(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("insert into test(name) values(?)");
        super.declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
}
```

```
@Test
public void testSqlUpdate() {
    SqlUpdate insert = new InsertUserModel(jdbcTemplate);
    insert.update("name5");

    String updateSql = "update test set name=? where name=?";
    SqlUpdate update = new SqlUpdate(jdbcTemplate.getDataSource(), updateSql, new int[]{T
    update.update("name6", "name5");
    String deleteSql = "delete from test where name=:name";

    SqlUpdate delete = new SqlUpdate(jdbcTemplate.getDataSource(), deleteSql, new int[]{T
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    delete.updateByNamedParam(paramMap);
}
```

InsertUserModel类实现类似于SqlQuery实现，用于执行数据库插入操作，SqlUpdate还提供一种更简洁的构造器SqlUpdate(DataSource ds, String sql, int[] types)，其中types用于指定占位符或命名参数类型；SqlUpdate还支持命名参数，使用updateByNamedParam方法来进行命名参数操作。

7.3.4 存储过程及函数

StoredProcedure用于支持存储过程及函数，该类的使用同样类似于SqlQuery。

StoredProcedure提供execute方法用于执行存储过程及函数。

一、**StoredProcedure**如何调用自定义函数：

```
@Test
public void testStoredProcedure1() {
    StoredProcedure lengthFunction = new HsqldbLengthFunction(jdbcTemplate);
    Map<String,Object> outValues = lengthFunction.execute("test");
    Assert.assertEquals(4, outValues.get("result"));
}
```

StoredProcedure使用非常简单，定义StoredProcedure实现HsqldbLengthFunction，并调用execute方法执行即可，接下来看一下HsqldbLengthFunction实现：

```
package cn.javass.spring.chapter7;
//省略import
public class HsqldbLengthFunction extends StoredProcedure {
    public HsqldbLengthFunction(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("FUNCTION_TEST");
        super.declareParameter(
          new SqlReturnResultSet("result", new ResultSetExtractor<Integer>() {
            @Override
            public Integer extractData(ResultSet rs) throws SQLException, DataAccessExcep
                while(rs.next()) {
                    return rs.getInt(1);
                }
                return 0;
            }
        }));
        super.declareParameter(new SqlParameter("str", Types.VARCHAR));
        compile();
    }
}
```

StoredProcedure自定义函数使用类似于SqlQuery，首先设置数据源或JdbcTemplate对象，其次定义自定义函数，然后使用declareParameter进行参数描述，最后调用compile（可选）编译自定义函数。

接下来看一下mysql自定义函数如何使用：

```
@Test
public void testStoredProcedure2() {
    JdbcTemplate mysqlJdbcTemplate = new JdbcTemplate(getMysqlDataSource());
    String createFunctionSql =
        "CREATE FUNCTION FUNCTION_TEST(str VARCHAR(100)) " +
        "returns INT return LENGTH(str)";
    String dropFunctionSql = "DROP FUNCTION IF EXISTS FUNCTION_TEST";
    mysqlJdbcTemplate.update(dropFunctionSql);
    mysqlJdbcTemplate.update(createFunctionSql);
    StoredProcedure lengthFunction = new MysqlLengthFunction(mysqlJdbcTemplate);
    Map<String,Object> outValues = lengthFunction.execute("test");
    Assert.assertEquals(4, outValues.get("result"));
}
```

MysqlLengthFunction自定义函数使用与HsqldbLengthFunction使用完全一样，只是内部实现稍有差别：

```
package cn.javass.spring.chapter7;
//省略import
public class MysqlLengthFunction extends StoredProcedure {
    public MysqlLengthFunction(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("FUNCTION_TEST");
        super.setFunction(true);
        super.declareParameter(new SqlOutParameter("result", Types.INTEGER));
        super.declareParameter(new SqlParameter("str", Types.VARCHAR));
        compile();
    }
}
```

MysqlLengthFunction与HsqldbLengthFunction实现不同的地方有两点：

- **setFunction(true)**：表示是自定义函数调用，即编译后的sql为{?= call …}形式；如果使用hsqldb不能设置为true，因为在hsqldb中{?= call …}和{call …}含义一样；
- **declareParameter(new SqlOutParameter("result", Types.INTEGER))**：将自定义函数返回值类型直接描述为Types.INTEGER；SqlOutParameter必须指定name，而不用使用SqlReturnResultSet首先获取结果集，然后再从结果集获取返回值，这是mysql与hsqldb的区别；

一、**StoredProcedure**如何调用存储过程：

```
@Test
public void testStoredProcedure3() {
    StoredProcedure procedure = new HsqldbTestProcedure(jdbcTemplate);
    Map<String,Object> outValues = procedure.execute("test");
    Assert.assertEquals(0, outValues.get("outId"));
    Assert.assertEquals("Hello,test", outValues.get("inOutName"));
}
```

StoredProcedure存储过程实现HsqldbTestProcedure调用与HsqldbLengthFunction调用完全一样，不同的是在实现时，参数描述稍有不同：

```
package cn.javass.spring.chapter7;
//省略import
public class HsqldbTestProcedure extends StoredProcedure {
    public HsqldbTestProcedure(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("PROCEDURE_TEST");
        super.declareParameter(new SqlInOutParameter("inOutName", Types.VARCHAR));
        super.declareParameter(new SqlOutParameter("outId", Types.INTEGER));
        compile();
    }
}
```

- **declareParameter**：使用SqlInOutParameter描述INOUT类型参数，使用SqlOutParameter描述OUT类型参数，必须按顺序定义，不能颠倒。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2491.html】

【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3【私塾在线原创】

7.4 Spring提供的其它帮助

7.4.1 SimpleJdbc方式

Spring JDBC抽象框架提供SimpleJdbcInsert和SimpleJdbcCall类，这两个类通过利用JDBC驱动提供的数据库元数据来简化JDBC操作。

1、SimpleJdbcInsert： 用于插入数据，根据数据库元数据进行插入数据，本类用于简化插入操作，提供三种类型方法：execute方法用于普通插入、executeAndReturnKey及executeAndReturnKeyHolder方法用于插入时获取主键值、executeBatch方法用于批处理。

```
@Test
public void testSimpleJdbcInsert() {
    SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    Map<String, Object> args = new HashMap<String, Object>();
    args.put("name", "name5");
    insert.compile();
    //1.普通插入
    insert.execute(args);
    Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test"));
    //2.插入时获取主键值
    insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    insert.setGeneratedKeyName("id");
    Number id = insert.executeAndReturnKey(args);
    Assert.assertEquals(1, id);
    //3.批处理
    insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    insert.setGeneratedKeyName("id");
    int[] updateCount = insert.executeBatch(new Map[] {args, args, args});
    Assert.assertEquals(1, updateCount[0]);
    Assert.assertEquals(5, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

- **new SimpleJdbcInsert(jdbcTemplate)** ： 首次通过DataSource对象或JdbcTemplate对象初始化SimpleJdbcInsert；
- **insert.withTableName("test")** ： 用于设置数据库表名；
- **args** ： 用于指定插入时列名及值，如本例中只有name列名，即编译后的sql类似于“insert into test(name) values(?)”；
- **insert.compile()** ： 可选的编译步骤，在调用执行方法时自动编译，编译后不能再对insert对象修改；
- 执行： execute方法用于执行普通插入；executeAndReturnKey用于执行并获取自动生成主键（注意是Number类型），必须首先通过setGeneratedKeyName设置主键然后才能获取，如果想获取复合主键请使用setGeneratedKeyNames描述主键然后通过

executeReturningKeyHolder获取复合主键KeyHolder对象；executeBatch用于批处理；

2、SimpleJdbcCall：用于调用存储过程及自定义函数，本类用于简化存储过程及自定义函数调用。

```
@Test
public void testSimpleJdbcCall1() {
    //此处用mysql,因为hsqldb调用自定义函数和存储过程一样
    SimpleJdbcCall call = new SimpleJdbcCall(getMysqlDataSource());
    call.withFunctionName("FUNCTION_TEST");
    call.declareParameters(new SqlOutParameter("result", Types.INTEGER));
    call.declareParameters(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outVlaues = call.execute("test");
    Assert.assertEquals(4, outVlaues.get("result"));
}
```

- **new SimpleJdbcCall(getMysqlDataSource())**：通过DataSource对象或JdbcTemplate对象初始化SimpleJdbcCall；
- **withFunctionName("FUNCTION_TEST")**：定义自定义函数名；自定义函数sql语句将被编译为类似于{?= call …}形式；
- **declareParameters**：描述参数类型，使用方式与StoredProcedure对象一样；
- 执行：调用execute方法执行自定义函数；

```
@Test
public void testSimpleJdbcCall2() {
    //调用hsqldb自定义函数得使用如下方式
    SimpleJdbcCall call = new SimpleJdbcCall(jdbcTemplate);
    call.withProcedureName("FUNCTION_TEST");
call.declareParameters(new SqlReturnResultSet("result",
new ResultSetExtractor<Integer>() {
        @Override
        public Integer extractData(ResultSet rs)
throws SQLException, DataAccessException {
          while(rs.next()) {
            return rs.getInt(1);
          }
          return 0;
    }}));
    call.declareParameters(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outVlaues = call.execute("test");
    Assert.assertEquals(4, outVlaues.get("result"));
}
```

调用hsqldb数据库自定义函数与调用mysql自定义函数完全不同，详见StoredProcedure中的解释。

```
@Test
public void testSimpleJdbcCall3() {
  SimpleJdbcCall call = new SimpleJdbcCall(jdbcTemplate);
  call.withProcedureName("PROCEDURE_TEST");
  call.declareParameters(new SqlInOutParameter("inOutName", Types.VARCHAR));
  call.declareParameters(new SqlOutParameter("outId", Types.INTEGER));
  SqlParameterSource params =
  new MapSqlParameterSource().addValue("inOutName", "test");
  Map<String, Object> outVlaues = call.execute(params);
  Assert.assertEquals("Hello,test", outVlaues.get("inOutName"));
  Assert.assertEquals(0, outVlaues.get("outId"));
}
```

与自定义函数调用不同的是使用withProcedureName来指定存储过程名字；其他参数描述等完全一样。

7.4.2 控制数据库连接

Spring JDBC通过DataSource控制数据库连接，即通过DataSource实现获取数据库连接。

Spring JDBC提供了一下DataSource实现：

- **DriverManagerDataSource** ：简单封装了DriverManager获取数据库连接；通过DriverManager的getConnection方法获取数据库连接；
- **SingleConnectionDataSource** ：内部封装了一个连接，该连接使用后不会关闭，且不能在多线程环境中使用，一般用于测试；
- **LazyConnectionDataSourceProxy** ：包装一个DataSource，用于延迟获取数据库连接，只有在真正创建Statement等时才获取连接，因此再说实际项目中最后使用该代理包装原始DataSource从而使得只有在真正需要连接时才去获取。

第三方提供的DataSource实现主要有C3P0、Proxool、DBCP等，这些实现都具有数据库连接池能力。

DataSourceUtils： Spring JDBC抽象框架内部都是通过它的getConnection(DataSource dataSource)方法获取数据库连接，releaseConnection(Connection con, DataSource dataSource) 用于释放数据库连接，DataSourceUtils用于支持Spring管理事务，只有使用DataSourceUtils获取的连接才具有Spring管理事务。

7.4.3 获取自动生成的主键

有许多数据库提供自动生成主键的能力，因此我们可能需要获取这些自动生成的主键，JDBC 3.0标准支持获取自动生成的主键，且必须数据库支持自动生成键获取。

1 ）JdbcTemplate 获取自动生成主键方式：

```
@Test
public void testFetchKey1() throws SQLException {
    final String insertSql = "insert into test(name) values('name5')";
    KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
    jdbcTemplate.update(new PreparedStatementCreator() {
        @Override
       public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            return conn.prepareStatement(insertSql, new String[]{"ID"});
      }}, generatedKeyHolder);
    Assert.assertEquals(0, generatedKeyHolder.getKey());
}
```

使用JdbcTemplate的update(final PreparedStatementCreator psc, final KeyHolder generatedKeyHolder)方法执行需要返回自动生成主键的插入语句，其中psc用于创建PreparedStatement并指定自动生成键，如“prepareStatement(insertSql, new String[]{"ID"})”；generatedKeyHolder是KeyHolder类型，用于获取自动生成的主键或复合主键；如使用getKey方法获取自动生成的主键。

2）SqlUpdate 获取自动生成主键方式：

```
@Test
public void testFetchKey2() {
    final String insertSql = "insert into test(name) values('name5')";
    KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
    SqlUpdate update = new SqlUpdate();
    update.setJdbcTemplate(jdbcTemplate);
    update.setReturnGeneratedKeys(true);
    //update.setGeneratedKeysColumnNames(new String[]{"ID"});
    update.setSql(insertSql);
    update.update(null, generatedKeyHolder);
    Assert.assertEquals(0, generatedKeyHolder.getKey());
}
```

SqlUpdate获取自动生成主键方式和JdbcTemplate完全一样，可以使用setReturnGeneratedKeys（true）表示要获取自动生成键；也可以使用setGeneratedKeysColumnNames指定自动生成键列名。

3）SimpleJdbcInsert ： 前边示例已介绍，此处就不演示了。

7.4.4 JDBC批量操作

JDBC批处理用于减少与数据库交互的次数来提升性能，Spring JDBC抽象框架通过封装批处理操作来简化批处理操作

1）JdbcTemplate 批处理： 支持普通的批处理及占位符批处理；

```
@Test
public void testBatchUpdate1() {
    String insertSql = "insert into test(name) values('name5')";
    String[] batchSql = new String[] {insertSql, insertSql};
    jdbcTemplate.batchUpdate(batchSql);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

直接调用batchUpdate方法执行需要批处理的语句即可。

```
@Test
public void testBatchUpdate2() {
    String insertSql = "insert into test(name) values(?)";
    final String[] batchValues = new String[] {"name5", "name6"};
    jdbcTemplate.batchUpdate(insertSql, new BatchPreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement ps, int i) throws SQLException {
            ps.setString(1, batchValues[i]);
        }
        @Override
        public int getBatchSize() {
            return batchValues.length;
        }
    });
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

JdbcTemplate还可以通过batchUpdate(String sql, final BatchPreparedStatementSetter pss)方法进行批处理，该方式使用预编译语句，然后通过BatchPreparedStatementSetter实现进行设值（setValues）及指定批处理大小（getBatchSize）。

2 ）NamedParameterJdbcTemplate 批处理： 支持命名参数批处理；

```
@Test
public void testBatchUpdate3() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate = new NamedParameterJdbcTemplat
    String insertSql = "insert into test(name) values(:myName)";
    UserModel model = new UserModel();
    model.setMyName("name5");
    SqlParameterSource[] params = SqlParameterSourceUtils.createBatch(new Object[] {model
    namedParameterJdbcTemplate.batchUpdate(insertSql, params);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

通过batchUpdate(String sql, SqlParameterSource[] batchArgs)方法进行命名参数批处理，batchArgs指定批处理数据集。SqlParameterSourceUtils.createBatch用于根据JavaBean对象或者Map创建相应的BeanPropertySqlParameterSource或MapSqlParameterSource。

3) SimpleJdbcTemplate 批处理： 已更简单的方式进行批处理；

```
@Test
public void testBatchUpdate4() {
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(?)";
    List<Object[]> params = new ArrayList<Object[]>();
    params.add(new Object[]{"name5"});
    params.add(new Object[]{"name5"});
    simpleJdbcTemplate.batchUpdate(insertSql, params);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

本示例使用batchUpdate(String sql, List<Object[]> batchArgs)方法完成占位符批处理，当然也支持命名参数批处理等。

4 ）SimpleJdbcInsert 批处理：

```
@Test
public void testBatchUpdate5() {
    SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    Map<String, Object> valueMap = new HashMap<String, Object>();
    valueMap.put("name", "name5");
    insert.executeBatch(new Map[] {valueMap, valueMap});
   Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

如代码所示，使用executeBatch(Map<String, Object>[] batch)方法执行批处理。

【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

7.5 集成Spring JDBC及最佳实践

大多数情况下Spring JDBC都是与IOC容器一起使用。通过配置方式使用Spring JDBC。

而且大部分时间都是使用JdbcTemplate类（或SimpleJdbcTemplate和NamedParameterJdbcTemplate）进行开发，即可能80%时间使用JdbcTemplate类，而只有20%时间使用其他类开发，符合**80/20**法则。

Spring JDBC通过实现DaoSupport来支持一致的数据库访问。

Spring JDBC提供如下DaoSupport实现：

- **JdbcDaoSupport**：用于支持一致的JdbcTemplate访问；
- **NamedParameterJdbcDaoSupport:**继承JdbcDaoSupport，同时提供NamedParameterJdbcTemplate访问；
- **SimpleJdbcDaoSupport**：继承JdbcDaoSupport，同时提供SimpleJdbcTemplate访问。

由于JdbcTemplate、NamedParameterJdbcTemplate、SimpleJdbcTemplate类使用DataSourceUtils获取及释放连接，而且连接是与线程绑定的，因此这些JDBC模板类是线程安全的，即JdbcTemplate对象可以在多线程中重用。

接下来看一下Spring JDBC框架的最佳实践：

1)首先定义**Dao**接口

```
package cn.javass.spring.chapter7.dao;
import cn.javass.spring.chapter7.UserModel;
public interface IUserDao {
    public void save(UserModel model);
    public int countAll();
}
```

2）定义**Dao**实现，此处是使用**Spring JDBC**实现：

```
package cn.javass.spring.chapter7.dao.jdbc;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
import cn.javass.spring.chapter7.UserModel;
import cn.javass.spring.chapter7.dao.IUserDao;
public class UserJdbcDaoImpl extends SimpleJdbcDaoSupport implements IUserDao {
  private static final String INSERT_SQL = "insert into test(name) values(:myName)";
  private static final String COUNT_ALL_SQL = "select count(*) from test";

  @Override
  public void save(UserModel model) {
      getSimpleJdbcTemplate().update(INSERT_SQL, new BeanPropertySqlParameterSource(model
  }
  @Override
  public int countAll() {
      return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
  }
}
```

此处注意首先Spring JDBC实现放在dao.jdbc包里，如果有hibernate实现就放在dao.hibernate包里；其次实现类命名如UserJdbcDaoImpl，即×××JdbcDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

3）进行资源配置（**resources/chapter7/applicationContext-resources.xml**）：

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <list>
          <value>classpath:chapter7/resources.properties</value>
      </list>
    </property>
</bean>
```

PropertyPlaceholderConfigurer用于替换配置元数据，如本示例中将对bean定义中的${…}占位符资源用“classpath:chapter7/resources.properties”中相应的元素替换。

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.LazyConnectionDataSource
    <property name="targetDataSource">
      <bean class="org.logicalcobwebs.proxool.ProxoolDataSource">
          <property name="driver" value="${db.driver.class}" />
          <property name="driverUrl" value="${db.url}" />
          <property name="user" value="${db.username}" />
          <property name="password" value="${db.password}" />
          <property name="maximumConnectionCount"
                 value="${proxool.maxConnCount}" />
          <property name="minimumConnectionCount"
                 value="${proxool.minConnCount}" />
          <property name="statistics" value="${proxool.statistics}" />
          <property name="simultaneousBuildThrottle"
                   value="${proxool.simultaneousBuildThrottle}" />
          <property name="trace" value="${proxool.trace}" />
      </bean>
    </property>
</bean>
```

dataSource定义数据源，本示例使用proxool数据库连接池，并使用LazyConnectionDataSourceProxy包装它，从而延迟获取数据库连接；${db.driver.class}将被“classpath:chapter7/resources.properties”中的“db.driver.class”元素属性值替换。

proxool数据库连接池：本示例使用proxool-0.9.1版本，请到proxool官网下载并添加proxool-0.9.1.jar和proxool-cglib.jar到类路径。

ProxoolDataSource属性含义如下：

- driver：指定数据库驱动；
- driverUrl：数据库连接；
- username：用户名；
- password：密码；
- maximumConnectionCount：连接池最大连接数量；
- minimumConnectionCount：连接池最小连接数量；
- statistics：连接池使用样本状况统计；如1m,15m,1h,1d表示没1分钟、15分钟、1小时及1天进行一次样本统计；
- simultaneousBuildThrottle：一次可以创建连接的最大数量；
- trace：true表示被执行的每个sql都将被记录（DEBUG级别时被打印到相应的日志文件）；

4）定义资源文件（**classpath:chapter7/resources.properties**）：

```
proxool.maxConnCount=10
proxool.minConnCount=5
proxool.statistics=1m,15m,1h,1d
proxool.simultaneousBuildThrottle=30
proxool.trace=false
db.driver.class=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:mem:test
db.username=sa
db.password=
```

用于替换配置元数据中相应的占位符数据，如${db.driver.class}将被替换为“org.hsqldb.jdbcDriver”。

5）**dao**定义配置（**chapter7/applicationContext-jdbc.xml**）：

```
<bean id="abstractDao" abstract="true">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="userDao"
     class="cn.javass.spring.chapter7.dao.jdbc.UserJdbcDaoImpl"
    parent="abstractDao"/>
```

首先定义抽象的abstractDao，其有一个dataSource属性，从而可以让继承的子类自动继承dataSource属性注入；然后定义userDao，且继承abstractDao，从而继承dataSource注入；我们在此给配置文件命名为applicationContext-jdbc.xml表示Spring JDBC DAO实现；如果使

用hibernate实现可以给配置文件命名为applicationContext-hibernate.xml。

6）最后测试一下吧（**cn.javass.spring.chapter7. JdbcTemplateTest**）：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter7/applicationContext-jdbc.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
     Assert.assertEquals(1, userDao.countAll());
}
```

首先读取配置文件，获取IUserDao接口实现，然后再调用IUserDao接口方法，进行数据库操作，这样对于开发人员使用来说，只面向接口，不关心实现，因此很容易更换实现，比如像更换为hibernate实现非常简单。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2493.html】

【第八章】 对ORM的支持 之 8.1 概述 ——跟我学spring3

8.1 概述

8.1.1 ORM框架

ORM全称对象关系映射（Object/Relation Mapping），指将Java对象状态自动映射到关系数据库中的数据上，从而提供透明化的持久化支持，即把一种形式转化为另一种形式。

对象与关系数据库之间是不匹配，我们把这种不匹配称为阻抗失配，主要表现在：

- 关系数据库首先不支持面向对象技术如继承、多态，如何使关系数据库支持它们；
- 关系数据库是由表来存放数据，而面向对象使用对象来存放状态；其中表的列称为属性，而对象的属性就是属性，因此需要通过解决这种不匹配；
- 如何将对象透明的持久化到关系数据库表中；
- 如果一个对象存在横跨多个表的数据，应该如何为对象建模和映射。

其中这些阻抗失配只是其中的一小部分，比如还有如何将SQL集合函数结果集映射到对象，如何在对象中处理主键等。

ORM框架就是用来解决这种阻抗失配，提供关系数据库的对象化支持。

ORM框架不是万能的，同样符合**80/20**法则，应解决的最核心问题是如何在关系数据库表中的行和对象进行映射，并自动持久化对象到关系数据库。

ORM解决方案适用于解决透明持久化、小结果集查询等；对于复杂查询，大结果集数据处理还是没有任何帮助的。

目前已经有许多ORM框架产生，如Hibernate、JDO、JPA、iBATIS等等，这些ORM框架各有特色，Spring对这些ORM框架提供了很好的支持，接下来首先让我们看一下Spring如何支持这些ORM框架。

8.1.2 Spring对ORM的支持

Spring对ORM的支持主要表现在以下方面：

- 一致的异常体系结构，对第三方ORM框架抛出的专有异常进行包装，从而在使我们在Spring中只看到DataAccessException异常体系；
- 一致的DAO抽象支持：提供类似与JdbcSupport的DAO支持类HibernateDaoSupport，使用HibernateTemplate模板类来简化常用操作，HibernateTemplate提供回调接口来支持复

杂操作；

- Spring事务管理：Spring对所有数据访问提供一致的事务管理，通过配置方式，简化事务管理。

Spring还在测试、数据源管理方面提供支持，从而允许方便测试，简化数据源使用。

接下来让我们学习一下Spring如何集成ORM框架—Hibernate。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2495.html】

【第八章】 对ORM的支持 之 8.2 集成Hibernate3 ——跟我学spring3

8.2 集成Hibernate3

Hibernate是全自动的ORM框架，能自动为对象生成相应SQL并透明的持久化对象到数据库。

Spring2.5+版本支持**Hibernate 3.1+**版本，不支持低版本，**Spring3.0.5**版本提供对**Hibernate 3.6.0 Final**版本支持。

8.2.1 如何集成

Spring通过使用如下Bean进行集成Hibernate：

- LocalSessionFactoryBean ：用于支持XML映射定义读取：

configLocation和configLocations：用于定义Hibernate配置文件位置，一般使用如classpath:hibernate.cfg.xml形式指定；

mappingLocations ：用于指定Hibenate映射文件位置，如chapter8/hbm/user.hbm.xml；

hibernateProperties：用于定义Hibernate属性，即Hibernate配置文件中的属性；

dataSource：定义数据源；

hibernateProperties、dataSource用于消除Hibernate配置文件，因此如果使用configLocations指定配置文件，就不要设置这两个属性了，否则会产生重复配置。推荐使用dataSource来指定数据源，而使用hibernateProperties指定Hibernate属性。

- AnnotationSessionFactoryBean：用于支持注解风格映射定义读取，该类继承LocalSessionFactoryBean并额外提供自动查找注解风格配置模型的能力：

annotatedClasses：设置注解了模型类，通过注解指定映射元数据。

packagesToScan：通过扫描指定的包获取注解模型类，而不是手工指定，如“cn.javass.**.model”将扫描cn.javass包及子包下的model包下的所有注解模型类。

接下来学习一下Spring如何集成Hibernate吧：

1、准备**jar**包：

首先准备Spring对ORM框架支持的jar包：

- org.springframework.orm-3.0.5.RELEASE.jar //提供对ORM框架集成

下载hibernate-distribution-3.6.0.Final包，获取如下Hibernate需要的jar包：

- hibernate3.jar //核心包
- lib\required\antlr-2.7.6.jar //HQL解析时使用的包
- lib\required\javassist-3.9.0.GA.jar //字节码类库，类似于cglib
- lib\required\commons-collections-3.1.jar //对集合类型支持包，前边测试时已经提供过了，无需再拷贝该包了
- lib\required\dom4j-1.6.1.jar //xml解析包，用于解析配置使用
- lib\required\jta-1.1.jar //JTA事务支持包
- lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar //用于支持JPA

下载slf4j-1.6.1.zip（http://www.slf4j.org/download.html），slf4j是日志系统门面（Simple Logging Facade for Java），用于对各种日志框架提供给一致的日志访问接口，从而能随时替换日志框架（如log4j、java.util.logging）：

- slf4j-api-1.6.1.jar //核心API
- slf4j-log4j12-1.6.1.jar //log4j实现

将这些jar包添加到类路径中。

2、对象模型定义，此处使用第七章中的**UserModel**：

```
package cn.javass.spring.chapter7;
public class UserModel {
    private int id;
    private String myName;
    //省略getter和setter
}
```

3、**Hibernate**映射定义（**chapter8/hbm/user.hbm.xml**），定义对象和数据库之间的映射：

```
&lt;?xml version="1.0" encoding="UTF-8"?&gt;
&lt;!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"&gt;
&lt;hibernate-mapping&gt;
    &lt;class name="cn.javass.spring.chapter7.UserModel" table="test"&gt;
        &lt;id name="id" column="id"&gt;&lt;generator class="native"/&gt;&lt;/id&gt;
        &lt;property name="myName" column="name"/&gt;
    &lt;/class&gt;
&lt;/hibernate-mapping&gt;
```

4、 数据源定义，此处使用第**7**章的配置文件，即**“chapter7/applicationContext-resources.xml”**文件。

5、 **SessionFactory**配置定义（**chapter8/applicationContext-hibernate.xml**）：

```
&lt;bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactor
    &lt;property name="dataSource" ref="dataSource"/&gt; &lt;!-- 指定数据源 --&gt;
      &lt;property name="mappingResources"&gt;     &lt;!-- 指定映射定义 --&gt;
        &lt;list&gt;
          &lt;value&gt;chapter8/hbm/user.hbm.xml&lt;/value&gt;
        &lt;/list&gt;
      &lt;/property&gt;
      &lt;property name="hibernateProperties"&gt;   &lt;!--指定Hibernate属性 --&gt;
        &lt;props&gt;
          &lt;prop key="hibernate.dialect"&gt;
              org.hibernate.dialect.HSQLDialect
          &lt;/prop&gt;
        &lt;/props&gt;
      &lt;/property&gt;
&lt;/bean&gt;
```

6、 获取**SessionFactory**：

```
package cn.javass.spring.chapter8;
//省略import
public class HibernateTest {
    private static SessionFactory sessionFactory;
    @BeforeClass
    public static void beforeClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-hibernate.xml"};
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
        sessionFactory = ctx.getBean("sessionFactory", SessionFactory.class);
}
}
```

此处我们使用了chapter7/applicationContext-resources.xml定义的“dataSource”数据源，通过ctx.getBean("sessionFactory", SessionFactory.class)获取SessionFactory。

7、通过**SessionFactory**获取**Session**对象进行创建和删除表：

```
@Before
public void setUp() {
  //id自增主键从0开始
  final String createTableSql = "create memory table test" + "(id int GENERATED BY DEFAUL
  sessionFactory.openSession().
  createSQLQuery(createTableSql).executeUpdate();
}
@After
public void tearDown() {
    final String dropTableSql = "drop table test";
    sessionFactory.openSession().
    createSQLQuery(dropTableSql).executeUpdate();
}
```

使用SessionFactory创建Session，然后通过Session对象的createSQLQuery创建本地SQL执行创建和删除表。

8、使用**SessionFactory**获取**Session**对象进行持久化数据：

```
@Test
public void testFirst() {
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    try {
        transaction = beginTransaction(session);
        UserModel model = new UserModel();
        model.setMyName("myName");
        session.save(model);
    } catch (RuntimeException e) {
        rollbackTransaction(transaction);
        throw e;
    } finally {
        commitTransaction(session);
    }
}
```

```
private Transaction beginTransaction(Session session) {
    Transaction transaction = session.beginTransaction();
    transaction.begin();
    return transaction;
}
private void rollbackTransaction(Transaction transaction) {
   if(transaction != null) {
        transaction.rollback();
    }
}
private void commitTransaction(Session session) {
    session.close();
}
```

使用SessionFactory获取Session进行操作，必须自己控制事务，而且还要保证各个步骤不会出错，有没有更好的解决方案把我们从编程事务中解脱出来？Spring提供了HibernateTemplate模板类用来简化事务处理和常见操作。

8.2.2 使用HibernateTemplate

HibernateTimplate模板类用于简化事务管理及常见操作，类似于JdbcTemplate模板类，对于复杂操作通过提供HibernateCallback回调接口来允许更复杂的操作。

接下来示例一下HibernateTemplate的使用：

```
@Test
public void testHibernateTemplate() {
HibernateTemplate hibernateTemplate =
new HibernateTemplate(sessionFactory);
    final UserModel model = new UserModel();
    model.setMyName("myName");
    hibernateTemplate.save(model);
    //通过回调允许更复杂操作
    hibernateTemplate.execute(new HibernateCallback&lt;Void&gt;() {
        @Override
        public Void doInHibernate(Session session)
            throws HibernateException, SQLException {
            session.save(model);
            return null;
        }});
}
```

通过new HibernateTemplate(sessionFactory) 创建HibernateTemplate模板类对象，通过调用模板类的save方法持久化对象，并且自动享受到Spring管理事务的好处。

而且HibernateTemplate 提供使用HibernateCallback回调接口的方法execute用来支持复杂操作，当然也自动享受到Spring管理事务的好处。

8.2.3 集成Hibernate及最佳实践

类似于JdbcDaoSupport类，Spring对Hibernate也提供了HibernateDaoSupport类来支持一致的数据库访问。HibernateDaoSupport也是DaoSupport实现：

接下来示例一下Spring集成Hibernate的最佳实践：

1、 定义**Dao**接口，此处使用**cn.javass.spring.chapter7.dao. IUserDao**：

2、 定义**Dao**接口实现，此处是**Hibernate**实现：

```
package cn.javass.spring.chapter8.dao.hibernate;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import cn.javass.spring.chapter7.UserModel;
import cn.javass.spring.chapter7.dao.IUserDao;
public class UserHibernateDaoImpl extends HibernateDaoSupport implements IUserDao {
    private static final String COUNT_ALL_HQL = "select count(*) from UserModel";
    @Override
    public void save(UserModel model) {
        getHibernateTemplate().save(model);
    }
    @Override
    public int countAll() {
        Number count = (Number) getHibernateTemplate().find(COUNT_ALL_HQL).get(0);
        return count.intValue();
    }
}
```

此处注意首先Hibernate实现放在dao.hibernate包里，其次实现类命名如UserHibernateDaoImpl，即×××HibernateDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

3、进行资源配置，使用**resources/chapter7/applicationContext-resources.xml**：

4、**dao**定义配置，在**chapter8/applicationContext-hibernate.xml**中添加如下配置：

```
&lt;bean id="abstractDao" abstract="true"&gt;
    &lt;property name="sessionFactory" ref="sessionFactory"/&gt;
&lt;/bean&gt;
&lt;bean id="userDao"  class="cn.javass.spring.chapter8.dao.hibernate.UserHibernateDaoImp
```

首先定义抽象的abstractDao，其有一个sessionFactory属性，从而可以让继承的子类自动继承sessionFactory属性注入；然后定义userDao，且继承abstractDao，从而继承sessionFactory注入；我们在此给配置文件命名为applicationContext-hibernate.xml表示

Hibernate实现。

5、 最后测试一下吧（**cn.javass.spring.chapter8. HibernateTest**）：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-hibernate.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和Spring JDBC框架的最佳实践完全一样，除了使用applicationContext-hibernate.xml代替了applicationContext-jdbc.xml，其他完全一样。也就是说，DAO层的实现替换可以透明化。

8.2.4 Spring+Hibernate的CRUD

Spring+Hibernate CRUD（增删改查）我们使用注解类来示例，让我们看具体示例吧：

1、首先定义带注解的模型对象**UserModel2**：

- 使用JPA注解@Table指定表名映射；
- 使用注解@Id指定主键映射；
- 使用注解@ Column指定数据库列映射；

```
package cn.javass.spring.chapter8;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "test")
public class UserModel2 {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(name = "name")
    private String myName;
    //省略getter和setter
}
```

2、 定义配置文件（**chapter8/applicationContext-hibernate2.xml**）：

2.1、 定义**SessionFactory**：

此处使用AnnotationSessionFactoryBean通过annotatedClasses属性指定注解模型来定义映射元数据；

```
&lt;bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.Annotat
&lt;property name="dataSource" ref="dataSource"/&gt;  &lt;!-- 1、指定数据源 --&gt;
  &lt;property name="annotatedClasses"&gt;           &lt;!-- 2、指定注解类 --&gt;
     &lt;list&gt;&lt;value&gt;cn.javass.spring.chapter8.UserModel2&lt;/value&gt;&lt;/list
&lt;/property&gt;
  &lt;property name="hibernateProperties"&gt;&lt;!-- 3、指定Hibernate属性 --&gt;
    &lt;props&gt;
     &lt;prop key="hibernate.dialect"&gt;
        org.hibernate.dialect.HSQLDialect
     &lt;/prop&gt;
    &lt;/props&gt;
  &lt;/property&gt;
&lt;/bean&gt;
```

2.2、定义HibernateTemplate ：

```
&lt;bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTempla
    &lt;property name="sessionFactory" ref="sessionFactory"/&gt;
&lt;/bean&gt;
```

3、最后进行CURD测试吧：

```
@Test
public void testCURD() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-hibernate2.xml"};
    ApplicationContext ctx =  new ClassPathXmlApplicationContext(configLocations);
    HibernateTemplate hibernateTemplate =  ctx.getBean(HibernateTemplate.class);
    UserModel2 model = new UserModel2();
    model.setMyName("test");
    insert(hibernateTemplate, model);
    select(hibernateTemplate, model);
    update(hibernateTemplate, model);
    delete(hibernateTemplate, model);
}

private void insert(HibernateTemplate hibernateTemplate, UserModel2 model) {
    hibernateTemplate.save(model);
}
private void select(HibernateTemplate hibernateTemplate, UserModel2 model) {
    UserModel2 model2 = hibernateTemplate.get(UserModel2.class, 0);
    Assert.assertEquals(model2.getMyName(), model.getMyName());
    List&lt;UserModel2&gt; list = hibernateTemplate.find("from UserModel2");
    Assert.assertEquals(list.get(0).getMyName(), model.getMyName());
}
private void update(HibernateTemplate hibernateTemplate, UserModel2 model) {
    model.setMyName("test2");
    hibernateTemplate.update(model);
}
private void delete(HibernateTemplate hibernateTemplate, UserModel2 model) {
    hibernateTemplate.delete(model);
}
```

Spring集成Hibernate进行增删改查是不是比Spring JDBC方式简单许多，而且支持注解方式配置映射元数据，从而减少映射定义配置文件数量。

【第八章】 对ORM的支持 之 8.3 集成iBATIS ——跟我学spring3

8.3 集成iBATIS

iBATIS是一个半自动化的ORM框架，需要通过配置方式指定映射SQL语句，而不是由框架本身生成（如Hibernate自动生成对应SQL来持久化对象），即Hibernate属于全自动ORM框架。

Spring提供对iBATIS 2.X的集成，提供一致的异常体系、一致的DAO访问支持、Spring管理事务支持。

Spring 2.5.5+版本支持**iBATIS 2.3+**版本，不支持低版本。

8.3.1 如何集成

Spring通过使用如下Bean进行集成iBATIS：

- SqlMapClientFactoryBean：用于集成iBATIS。

configLocation和configLocations：用于指定SQL Map XML配置文件，用于指定如数据源等配置信息；

mappingLocations：用于指定SQL Map映射文件，即半自动概念中的SQL语句定义；

sqlMapClientProperties：定义iBATIS 配置文件配置信息；

dataSource：定义数据源。

如果在Spring配置文件中指定了DataSource，就不要在iBATIS配置文件指定了，否则Spring配置文件指定的DataSource将覆盖iBATIS配置文件中定义的DataSource。

接下来示例一下如何集成iBATIS：

1、准备需要的**jar**包，从**spring-framework-3.0.5.RELEASE-dependencies.zip**中拷贝如下**jar**包：

com.springsource.com.ibatis-2.3.4.726.jar

2、 对象模型定义，此处使用第七章中的**UserModel**；

3、 iBATIS映射定义（**chapter8/sqlmaps/UserSQL.xml**）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="UserSQL">
  <statement id="createTable">
    <!--id自增主键从0开始 -->
    <![CDATA[
        create memory table test(
          id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
          name varchar(100))
    ]]>
  </statement>
  <statement id="dropTable">
    <![CDATA[  drop table test  ]]>
  </statement>
  <insert id="insert" parameterClass="cn.javass.spring.chapter7.UserModel">
    <![CDATA[
      insert into test(name) values (#myName#)
    ]]>
    <selectKey resultClass="int" keyProperty="id" type="post">
      <!-- 获取hsqldb插入的主键 -->
      call identity();
      <!-- mysql使用select last_insert_id();获取插入的主键 -->
    </selectKey>
  </insert>
</sqlMap>
```

4、 iBATIS配置文件（chapter8/sql-map-config.xml）定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
    <settings enhancementEnabled="true" useStatementNamespaces="true"
        maxTransactions="20" maxRequests="32" maxSessions="10"/>
    <sqlMap resource="chapter8/sqlmaps/UserSQL.xml"/>
</sqlMapConfig>
```

5、 数据源定义，此处使用第7章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

6、 SqlMapClient配置（chapter8/applicationContext-ibatis.xml）定义：

```
<bean id="sqlMapClient"
    class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <!-- 1、指定数据源 -->
    <property name="dataSource" ref="dataSource"/>
    <!-- 2、指定配置文件 -->
    <property name="configLocation" value="chapter8/sql-map-config.xml"/>
</bean>
```

7、 获取SqlMapClient：

```
package cn.javass.spring.chapter8;
//省略import
public class IbatisTest {
    private static SqlMapClient sqlMapClient;
    @BeforeClass
    public static void setUpClass() {
      String[] configLocations = new String[] {
          "classpath:chapter7/applicationContext-resources.xml",
          "classpath:chapter8/applicationContext-ibatis.xml"};
      ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
      sqlMapClient = ctx.getBean(SqlMapClient.class);
  }
}
```

此处我们使用了chapter7/applicationContext-resources.xml定义的“dataSource”数据源，通过ctx.getBean(SqlMapClient.class)获取SqlMapClient。

8、通过**SqlMapClient**创建和删除表：

```
@Before
public void setUp() throws SQLException {
    sqlMapClient.update("UserSQL.createTable");
}
@After
public void tearDown() throws SQLException {
    sqlMapClient.update("UserSQL.dropTable");
}
```

9、使用**SqlMapClient**进行对象持久化：

```
@Test
public void testFirst() throws SQLException {
    UserModel model = new UserModel();
    model.setMyName("test");
    SqlMapSession session = null;
    try {
        session = sqlMapClient.openSession();
        beginTransaction(session);
        session.insert("UserSQL.insert", model);
        commitTransaction(session);
    } catch (SQLException e) {
        rollbackTransacrion(session);
        throw e;
    } finally {
        closeSession(session);
    }
}
private void closeSession(SqlMapSession session) {
   session.close();
}
private void rollbackTransacrion(SqlMapSession session) throws SQLException {
    if(session != null) {
        session.endTransaction();
    }
}
private void commitTransaction(SqlMapSession session) throws SQLException {
    session.commitTransaction();
}
private void beginTransaction(SqlMapSession session) throws SQLException {
    session.startTransaction();
}
```

同样令人心烦的事务管理和冗长代码，Spring通用提供了SqlMapClientTemplate模板类来解决这些问题。

8.3.2 使用 SqlMapClientTemplate

SqlMapClientTemplate模板类同样用于简化事务管理及常见操作，类似于JdbcTemplate模板类，对于复杂操作通过提供SqlMapClientCallback回调接口来允许更复杂的操作。

接下来示例一下SqlMapClientTemplate的使用：

```
@Test
public void testSqlMapClientTemplate() {
SqlMapClientTemplate sqlMapClientTemplate =
new SqlMapClientTemplate(sqlMapClient);
    final UserModel model = new UserModel();
    model.setMyName("myName");
    sqlMapClientTemplate.insert("UserSQL.insert", model);
    //通过回调允许更复杂操作
    sqlMapClientTemplate.execute(new SqlMapClientCallback<Void>() {
        @Override
        public Void doInSqlMapClient(SqlMapExecutor session) throws SQLException {
            session.insert("UserSQL.insert", model);
            return null;
    }});
}
```

通过new SqlMapClientTemplate(sqlMapClient)创建HibernateTemplate模板类对象，通过调用模板类的save方法持久化对象，并且自动享受到Spring管理事务的好处。

而且SqlMapClientTemplate提供使用SqlMapClientCallback回调接口的方法execute用来支持复杂操作，当然也自动享受到Spring管理事务的好处。

8.3.3集成iBATIS及最佳实践

类似于JdbcDaoSupport类，Spring对iBATIS也提供了SqlMapClientDaoSupport类来支持一致的数据库访问。SqlMapClientDaoSupport也是DaoSupport实现：

接下来示例一下Spring集成iBATIS的最佳实践：

1、 定义**Dao**接口，此处使用**cn.javass.spring.chapter7.dao.IUserDao**：

2、 定义**Dao**接口实现，此处是**iBATIS**实现：

```
package cn.javass.spring.chapter8.dao.ibatis;
//省略import
public class UserIbatisDaoImpl extends SqlMapClientDaoSupport
    implements IUserDao {
    @Override
    public void save(UserModel model) {
        getSqlMapClientTemplate().insert("UserSQL.insert", model);
    }
    @Override
    public int countAll() {
        return (Integer) getSqlMapClientTemplate().queryForObject("UserSQL.countAll");
    }
}
```

3、修改**iBATS**映射文件（**chapter8/sqlmaps/UserSQL.xml**），添加**countAll**查询：

```
<select id="countAll" resultClass="java.lang.Integer">
    <![CDATA[ select count(*) from test ]]>
</select>
```

此处注意首先iBATIS实现放在dao.ibaitis包里，其次实现类命名如UserIbatisDaoImpl，即×××IbatisDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

4、进行资源配置，使用**resources/chapter7/applicationContext-resources.xml**：

5、**dao**定义配置，在**chapter8/applicationContext-ibatis.xml**中添加如下配置：

```
<bean id="abstractDao" abstract="true">
    <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
<bean id="userDao"
      class="cn.javass.spring.chapter8.dao.ibatis.UserIbatisDaoImpl"
      parent="abstractDao"/>
```

首先定义抽象的abstractDao，其有一个sqlMapClient属性，从而可以让继承的子类自动继承sqlMapClient属性注入；然后定义userDao，且继承abstractDao，从而继承sqlMapClient注入；我们在此给配置文件命名为applicationContext-ibatis.xml表示iBAITIS实现。

5、 最后测试一下吧（**cn.javass.spring.chapter8. IbatisTest**）：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-ibatis.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和Spring JDBC框架的最佳实践完全一样，除了使用applicationContext-ibatis.xml代替了applicationContext-jdbc.xml，其他完全一样。也就是说，DAO层的实现替换可以透明化。

8.3.4Spring+iBATIS的CURD

Spring集成iBATIS进行CURD（增删改查），也非常简单，首先配置映射文件，然后调用SqlMapClientTemplate相应的函数进行操作即可，此处就不介绍了。

8.3.5集成MyBatis及最佳实践

(本笔记写于2010年底)

2010年4月份 iBATIS团队发布iBATIS 3.0的GA版本的候选版本，在iBATIS 3中引入了泛型、注解支持等，因此需要Java5+才能使用，但在2010年6月16日，iBATIS团队决定从apache迁出并迁移到Google Code，并更名为MyBatis。目前新网站上文档并不完善。

目前iBATIS 2.x和MyBatis 3不是100%兼容的，如配置文件的DTD变更，SqlMapClient直接由SqlSessionFactory代替了，包名也有com.ibatis变成org.ibatis等等。

ibatis 3.x和MyBatis是兼容的，只需要将DTD变更一下就可以了。

感兴趣的朋友可以到http://www.mybatis.org/官网去下载最新的文档学习，作者只使用过iBATIS2.3.4及以前版本，没在新项目使用过最新的iBATIS 3.x和Mybatis，因此如果读者需要在项目中使用最新的MyBatis，请先做好调研再使用。

接下来示例一下Spring集成MyBatis的最佳实践：

1、准备需要的**jar**包，到**MyBatis**官网下载**mybatis 3.0.4**版本和**mybatis-spring 1.0.0**版本，并拷贝如下**jar**包到类路径：

- **mybatis-3.0.4\mybatis-3.0.4.jar //**核心**MyBatis**包
- **mybatis-spring-1.0.0\mybatis-spring-1.0.0.jar //**集成**Spring**包

2、对象模型定义，此处使用第七章中的**UserModel**；

3、**MyBatis**映射定义（**chapter8/sqlmaps/UserSQL-mybatis.xml**）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="UserSQL">
    <sql id="createTable">
    <!--id自增主键从0开始 -->
    <![CDATA[
      create memory table test(
        id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
        name varchar(100))
    ]]>
    </sql>
    <sql id="dropTable">
    <![CDATA[ drop table test ]]>
    </sql>
    <insert id="insert" parameterType="cn.javass.spring.chapter7.UserModel">
    <![CDATA[ insert into test(name) values (#{myName}) ]]>
        <selectKey resultType="int" keyProperty="id" order="AFTER">
            <!-- 获取hsqldb插入的主键 -->
            call identity();
            <!-- mysql使用select last_insert_id();获取插入的主键 -->
        </selectKey>
    </insert>
    <select id="countAll" resultType="java.lang.Integer">
    <![CDATA[ select count(*) from test ]]>
    </select>
</mapper>
```

从映射定义中可以看出MyBatis与iBATIS2.3.4有如下不同：

- http://ibatis.apache.org/dtd/sql-map-2.dtd 废弃，而使用http://mybatis.org/dtd/mybatis-3-mapper.dtd。
- <sqlMap>废弃，而使用<mapper>标签；
- <statement>废弃了，而使用<sql>标签；
- parameterClass属性废弃，而使用parameterType属性；
- resultClass属性废弃，而使用resultType属性；

• myName#方式指定命名参数废弃，而使用#{myName}方式。

3、MyBatis配置文件（chapter8/sql-map-config-mybatis.xml）定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <settings>
        <setting name="cacheEnabled" value="false"/>
    </settings>
    <mappers>
        <mapper resource="chapter8/sqlmaps/UserSQL-mybatis.xml"/>
    </mappers>
</configuration>
```

从配置定义中可以看出MyBatis与iBATIS2.3.4有如下不同：

- http://ibatis.apache.org/dtd/sql-map-config-2.dtd废弃，而使用http://mybatis.org/dtd/mybatis-3-config.dtd；
- < sqlMapConfig >废弃，而使用<configuration>；
- settings属性配置方式废弃，而改用子标签< setting name=".." value=".."/>方式指定属性，且一些属性被废弃，如maxTransactions；
- < sqlMap>废弃，而采用<mappers>标签及其子标签<mapper>定义。

4、定义**Dao**接口，此处使用**cn.javass.spring.chapter7.dao. IUserDao**：

5、定义**Dao**接口实现，此处是**MyBatis**实现：

```
package cn.javass.spring.chapter8.dao.mybatis;
//省略import
public class UserMybatisDaoImpl extends SqlSessionDaoSupport
implements IUserDao {
    @Override
    public void save(UserModel model) {
        getSqlSession().insert("UserSQL.insert", model);
    }
    @Override
    public int countAll() {
        return (Integer) getSqlSession().selectOne("UserSQL.countAll");
    }
}
```

和Ibatis集成方式不同的有如下地方：

- 使用SqlSessionDaoSupport来支持一致性的DAO访问，该类位于org.mybatis.spring.support包中，非Spring提供；
- 使用getSqlSession方法获取SqlSessionTemplate，在较早版本中是getSqlSessionTemplate方法名，不知为什么改成getSqlSession方法名，因此这个地方在使用时需要注意。
- SqlSessionTemplate是SqlSession接口的实现，并且自动享受Spring管理事务好处，因此从此处可以推断出为什么把获取模板类的方法名改为getSqlSession而不是getSqlSessionTemplate。

6、进行资源配置，使用**resources/chapter7/applicationContext-resources.xml**：

7、**dao**定义配置，在**chapter8/applicationContext-mybatis.xml**中添加如下配置：

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/><!-- 1、指定数据源 -->
  <property name="configLocation" value="chapter8/sql-map-config-mybatis.xml"/>
</bean>
<bean id="abstractDao" abstract="true">
   <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
<bean id="userDao"
           class="cn.javass.spring.chapter8.dao.mybatis.UserMybatisDaoImpl"
           parent="abstractDao"/>
```

和Ibatis集成方式不同的有如下地方：

- SqlMapClient类废弃，而使用SqlSessionFactory代替；
- 使用SqlSessionFactoryBean进行集成MyBatis。

首先定义抽象的abstractDao，其有一个sqlSessionFactory属性，从而可以让继承的子类自动继承sqlSessionFactory属性注入；然后定义userDao，且继承abstractDao，从而继承sqlSessionFactory注入；我们在此给配置文件命名为applicationContext-mybatis.xml表示MyBatis实现。

8、最后测试一下吧（**cn.javass.spring.chapter8. IbatisTest**）：

```
@Test
public void testMybatisBestPractice() {
    String[] configLocations = new String[] {
        "classpath:chapter7/applicationContext-resources.xml",
        "classpath:chapter8/applicationContext-mybatis.xml"};
     ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和Spring 集成Ibatis的最佳实践完全一样，除了使用applicationContext-mybatis.xml代替了applicationContext-ibatis.xml，其他完全一样，且MyBatis 3.x与Spring整合只能运行在Spring3.x。

在写本书时，MyBatis与Spring集成所定义的API不稳定，且期待Spring能在发布新版本时将加入对MyBatis的支持。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2498.html】

【第八章】 对ORM的支持 之 8.4 集成JPA ——跟我学spring3

8.4 集成JPA

JPA全称为Java持久性API（Java Persistence API），JPA是Java EE 5标准之一，是一个ORM规范，由厂商来实现该规范，目前有Hibernate、OpenJPA、TopLink、EclipseJPA等实现。

8.4.1 如何集成

Spring目前提供集成Hibernate、OpenJPA、TopLink、EclipseJPA四个JPA标准实现。

Spring通过使用如下Bean进行集成JPA（EntityManagerFactory）：

- **LocalEntityManagerFactoryBean**：适用于那些仅使用JPA进行数据访问的项目，该FactoryBean将根据JPA PersistenceProvider自动检测配置文件进行工作，一般从“META-INF/persistence.xml”读取配置信息，这种方式最简单，但不能设置Spring中定义的DataSource，且不支持Spring管理的全局事务，而且JPA 实现商可能在JVM启动时依赖于VM agent从而允许它们进行持久化类字节码转换（不同的实现厂商要求不同，需要时阅读其文档），不建议使用这种方式；

persistenceUnitName：指定持久化单元的名称；

使用方式：

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFact
    <property name="persistenceUnitName" value="persistenceUnit"/>
</bean>
```

- 从**JNDI**中获取：用于从Java EE服务器获取指定的EntityManagerFactory，这种方式在进行Spring事务管理时一般要使用JTA事务管理；

使用方式：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">
  <jee:jndi-lookup id="entityManagerFactory"  jndi-name="persistence/persistenceUnit"/>
</beans>
```

此处需要使用“jee”命名标签，且使用<jee:jndi-lookup>标签进行JNDI查找，“jndi-name”属性用于指定JNDI名字。

- **LocalContainerEntityManagerFactoryBean**：适用于所有环境的FactoryBean，能全面控制EntityManagerFactory配置,如指定Spring定义的DataSource等等。

persistenceUnitManager：用于获取JPA持久化单元，默认实现DefaultPersistenceUnitManager用于解决多配置文件情况

dataSource：用于指定Spring定义的数据源；

persistenceXmlLocation：用于指定JPA配置文件，对于对配置文件情况请选择设置persistenceUnitManager属性来解决；

persistenceUnitName：用于指定持久化单元名字；

persistenceProvider：用于指定持久化实现厂商类；如Hibernate为org.hibernate.ejb.HibernatePersistence类；

jpaVendorAdapter：用于设置实现厂商JPA实现的特定属性，如设置Hibernate的是否自动生成DDL的属性generateDdl；这些属性是厂商特定的，因此最好在这里设置；目前Spring提供HibernateJpaVendorAdapter、OpenJpaVendorAdapter、EclipseLinkJpaVendorAdapter、TopLinkJpaVendorAdapter、OpenJpaVendorAdapter四个实现。其中最重要的属性是“**database**”，用来指定使用的数据库类型，从而能根据数据库类型来决定比如如何将数据库特定异常转换为Spring的一致性异常，目前支持如下数据库（**DB2**、**DERBY**、**H2**、**HSQL**、**INFORMIX**、**MYSQL**、**ORACLE**、**POSTGRESQL**、**SQL_SERVER**、**SYBASE**）。

jpaDialect：用于指定一些高级特性，如事务管理，获取具有事务功能的连接对象等，目前Spring提供HibernateJpaDialect、OpenJpaDialect 、EclipseLinkJpaDialect、TopLinkJpaDialect、和DefaultJpaDialect实现，注意DefaultJpaDialect不提供任何功能，因此在使用特定实现厂商JPA实现时需要指定JpaDialect实现，如使用Hibernate就使用HibernateJpaDialect。当指定**jpaVendorAdapter**属性时可以不指定**jpaDialect**，会自动设置相应的**JpaDialect**实现；

jpaProperties和**jpaPropertyMap**：指定JPA属性；如Hibernate中指定是否显示SQL的“hibernate.show_sql”属性，对于jpaProperties设置的属性自动会放进jpaPropertyMap中；

loadTimeWeaver：用于指定LoadTimeWeaver实现，从而允许JPA 加载时修改相应的类文件。具体使用得参考相应的JPA规范实现厂商文档，如Hibernate就不需要指定loadTimeWeaver。

接下来学习一下Spring如何集成JPA吧：

1、准备**jar**包，从下载的**hibernate-distribution-3.6.0.Final**包中获取如下**Hibernate**需要的**jar**包从而支持**JPA**：

- lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar //用于支持JPA

2、对象模型定义，此处使用**UserModel2**：

```
package cn.javass.spring.chapter8;
//省略import
@Entity
@Table(name = "test")
public class UserModel2 {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(name = "name")
    private String myName;
    //省略getter和setter
}
```

注意此处使用的所有注解都是位于javax.persistence包下，如使用@org.hibernate.annotations.Entity 而非@javax.persistence. Entity将导致JPA不能正常工作。

1、 **JPA**配置定义（**chapter8/persistence.xml**），定义对象和数据库之间的映射：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence                      http:
    <persistence-unit name="persistenceUnit" transaction-type="RESOURCE_LOCAL"/>
</persistence>
```

在JPA配置文件中，我们指定要持久化单元名字，和事务类型，其他都将在Spring中配置。

2、 数据源定义，此处使用第**7**章的配置文件，即**“chapter7/applicationContext-resources.xml”**文件。

3、 **EntityManagerFactory**配置定义（**chapter8/applicationContext-jpa.xml**）：

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntity
  <property name="dataSource" ref="dataSource"/>
  <property name="persistenceXmlLocation" value="chapter8/persistence.xml"/>
  <property name="persistenceUnitName" value="persistenceUnit"/>
  <property name="persistenceProvider" ref="persistenceProvider"/>
  <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
  <property name="jpaDialect" ref="jpaDialect"/>
  <property name="jpaProperties">
      <props>
          <prop key="hibernate.show_sql">true</prop>
      </props>
  </property>
</bean>
<bean id="persistenceProvider" class="org.hibernate.ejb.HibernatePersistence"/>
```

```
<bean id="jpaVendorAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorA
   <property name="generateDdl" value="false" />
   <property name="database" value="HSQL"/>
</bean>
<bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
```

- **LocalContainerEntityManagerFactoryBean:**指定使用本地容器管理EntityManagerFactory，从而进行细粒度控制；
- **dataSource**属性指定使用Spring定义的数据源；
- **persistenceXmlLocation**指定JPA配置文件为chapter8/persistence.xml，且该配置文件非常简单，具体配置完全在Spring中进行；
- **persistenceUnitName**指定持久化单元名字，即JPA配置文件中指定的;
- **persistenceProvider**:指定JPA持久化提供商，此处使用Hibernate实现HibernatePersistence类；
- **jpaVendorAdapter**：指定实现厂商专用特性，即generateDdl= false表示不自动生成DDL，database= HSQL表示使用hsqldb数据库；
- **jpaDialect**：如果指定jpaVendorAdapter此属性可选，此处为HibernateJpaDialect；
- **jpaProperties**：此处指定“hibernate.show_sql =true”表示在日志系统debug级别下将打印所有生成的SQL。

4、 获取**EntityManagerFactory**：

```
package cn.javass.spring.chapter8;
//省略import
public class JPATest {
    private static EntityManagerFactory entityManagerFactory;
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-jpa.xml"};
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
        entityManagerFactory = ctx.getBean(EntityManagerFactory.class);
    }
}
```

此处我们使用了chapter7/applicationContext-resources.xml定义的“dataSource”数据源，通过ctx.getBean(EntityManagerFactory.class)获取EntityManagerFactory。

5、 通过EntityManagerFactory获取EntityManager进行创建和删除表：

```
@Before
public void setUp() throws SQLException {
   //id自增主键从0开始
   String createTableSql = "create memory table test" + "(id int GENERATED BY DEFAULT AS
   executeSql(createTableSql);
}
@After
public void tearDown() throws SQLException {
    String dropTableSql = "drop table test";
    executeSql(dropTableSql);
}

private void executeSql(String sql) throws SQLException {
  EntityManager em = entityManagerFactory.createEntityManager();
  beginTransaction(em);
  em.createNativeQuery(sql).executeUpdate();
  commitTransaction(em);
  closeEntityManager(em);
}
private void closeEntityManager(EntityManager em) {
  em.close();
}
private void rollbackTransacrion(EntityManager em) throws SQLException {
  if(em != null) {
     em.getTransaction().rollback();
  }
}
private void commitTransaction(EntityManager em) throws SQLException {
   em.getTransaction().commit();
}
private void beginTransaction(EntityManager em) throws SQLException {
   em.getTransaction().begin();
}
```

使用EntityManagerFactory创建EntityManager，然后通过EntityManager对象的createNativeQuery创建本地SQL执行创建和删除表。

6、 使用**EntityManagerFactory**获取**EntityManager**对象进行持久化数据：

```
@Test
public void testFirst() throws SQLException {
    UserModel2 model = new UserModel2();
    model.setMyName("test");
    EntityManager em = null;
    try {
        em = entityManagerFactory.createEntityManager();
        beginTransaction(em);
        em.persist(model);
        commitTransaction(em);
    } catch (SQLException e) {
        rollbackTransacrion(em);
        throw e;
    } finally {
      closeEntityManager(em);
    }
}
```

使用**EntityManagerFactory**获取**EntityManager**进行操作，看到这还能忍受冗长的代码和事务管理吗？Spring同样提供JpaTemplate模板类来简化这些操作。

大家有没有注意到此处的模型对象能自动映射到数据库，这是因为Hibernate JPA实现默认自动扫描类路径中的@Entity注解类及*.hbm.xml映射文件，可以通过更改Hibernate JPA属性“hibernate.ejb.resource_scanner”，并指定org.hibernate.ejb.packaging.Scanner接口实现来定制新的扫描策略。

8.4.2 使用JpaTemplate

JpaTemplate模板类用于简化事务管理及常见操作，类似于JdbcTemplate模板类，对于复杂操作通过提供JpaCallback回调接口来允许更复杂的操作。

接下来示例一下JpaTemplate的使用：

1、修改**Spring**配置文件（**chapter8/applicationContext-jpa.xml**），添加**JPA**事务管理器：

```
<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

- txManager：指定事务管理器，JPA使用JpaTransactionManager事务管理器实现，通过entityManagerFactory指定EntityManagerFactory；用于支持Java SE环境的**JPA**扩展的持久化上下文（**EXTENDED Persistence Context**）。

2、修改**JPATest**类，添加类变量**ctx**，用于后边使用其获取事务管理器使用：

```
package cn.javass.spring.chapter8;
public class JPATest {
    private static EntityManagerFactory entityManagerFactory;
    private static ApplicationContext ctx;
    @BeforeClass
    public static void beforeClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-jpa.xml"};
        ctx = new ClassPathXmlApplicationContext(configLocations);
        entityManagerFactory = ctx.getBean(EntityManagerFactory.class);
}
}
```

3）**JpaTemplate**模板类使用：

```
@Test
public void testJpaTemplate() {
final JpaTemplate jpaTemplate = new JpaTemplate(entityManagerFactory);
  final UserModel2 model = new UserModel2();
  model.setMyName("test1");
  PlatformTransactionManager txManager = ctx.getBean(PlatformTransactionManager.class);
  new TransactionTemplate(txManager).execute(
    new TransactionCallback<Void>() {
      @Override
      public Void doInTransaction(TransactionStatus status) {
        jpaTemplate.persist(model);
        return null;
      }
  });
  String COUNT_ALL = "select count(*) from UserModel";
  Number count = (Number) jpaTemplate.find(COUNT_ALL).get(0);
  Assert.assertEquals(1, count.intValue());
}
```

- **jpaTemplate**：可通过new JpaTemplate(entityManagerFactory)方式创建；
- **txManager**：通过ctx.getBean(PlatformTransactionManager.class)获取事务管理器；
- **TransactionTemplate**：通过new TransactionTemplate(txManager)创建事务模板对象，并通过execute方法执行TransactionCallback回调中的doInTransaction方法中定义需要执行的操作，从而将由模板类通过txManager事务管理器来进行事务管理，此处是调用jpaTemplate对象的persist方法进行持久化；
- **jpaTemplate.persist()**：根据JPA规范，在JPA扩展的持久化上下文，该操作必须运行在事务环境，还有persist()、 merge()、remove()操作也必须运行在事务环境；
- **jpaTemplate.find()**：根据JPA规范，该操作无需运行在事务环境，还有find()、getReference()、 refresh()、detach()和查询操作都无需运行在事务环境。

此实例与Hibernate和Ibatis有所区别，通过JpaTemplate模板类进行如持久化等操作时必须有运行在事务环境中，否则可能抛出如下异常或警告：

- **“javax.persistence.TransactionRequiredException：Executing an update/delete query”**：表示没有事务支持，不能执行更新或删除操作；
- 警告**“delaying identity-insert due to no transaction in progress”**：需要在日志系统启动debug模式才能看到，表示在无事务环境中无法进行持久化，而选择了延迟标识插入。

以上异常和警告是没有事务造成的，也是最让人困惑的问题，需要大家注意。

8.4.3 集成JPA及最佳实践

类似于JdbcDaoSupport类，Spring对JPA也提供了JpaDaoSupport类来支持一致的数据库访问。JpaDaoSupport也是DaoSupport实现：

接下来示例一下Spring集成JPA的最佳实践：

1、 定义**Dao**接口，此处使用**cn.javass.spring.chapter7.dao. IUserDao**：

2、 定义**Dao**接口实现，此处是**JPA**实现：

```
package cn.javass.spring.chapter8.dao.jpa;
//省略import
@Transactional(propagation = Propagation.REQUIRED)
public class UserJpaDaoImpl extends JpaDaoSupport implements IUserDao {
    private static final String COUNT_ALL_JPAQL = "select count(*) from UserModel";
    @Override
    public void save(UserModel model) {
        getJpaTemplate().persist(model);
    }
    @Override
    public int countAll() {
        Number count =
           (Number) getJpaTemplate().find(COUNT_ALL_JPAQL).get(0);
        return count.intValue();
    }
}
```

此处注意首先JPA实现放在dao.jpa包里，其次实现类命名如UserJpaDaoImpl，即×××JpaDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

另外在类上添加了**@Transactional**注解表示该类的所有方法将在调用时需要事务支持，propagation传播属性为Propagation.REQUIRED表示事务是必需的，如果执行该类的方法没有开启事务，将开启一个新的事务。

3、进行资源配置，使用**resources/chapter7/applicationContext-resources.xml**：

4、**dao**定义配置，在**chapter8/applicationContext-jpa.xml**中添加如下配置：

4.1、首先添加tx命名空间用于支持事务：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

4.2、为**@Transactional**注解事务开启事务支持：

```
<tx:annotation-driven transaction-manager="txManager"/>
```

只为类添加**@Transactional** 注解是不能支持事务的，需要通过<tx:annotation-driven>标签来开启事务支持，其中txManager属性指定事务管理器。

4.3、配置DAO Bean：

```
<bean id="abstractDao" abstract="true">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<bean id="userDao"
      class="cn.javass.spring.chapter8.dao.jpa.UserJpaDaoImpl"
      parent="abstractDao"/>
```

首先定义抽象的abstractDao，其有一个entityManagerFactory属性，从而可以让继承的子类自动继承entityManagerFactory属性注入；然后定义userDao，且继承abstractDao，从而继承entityManagerFactory注入；我们在此给配置文件命名为applicationContext-jpa.xml表示JPA实现。

5、最后测试一下吧（**cn.javass.spring.chapter8. JPATest**）：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-jpa.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和Spring JDBC框架的最佳实践完全一样，除了使用applicationContext-jpa.xml代替了applicationContext-jdbc.xml，其他完全一样。也就是说，DAO层的实现替换可以透明化。

还有与集成其他ORM框架不同的是JPA在进行持久化或更新数据库操作时需要事务支持。

8.4.4 Spring+JPA的CRUD

Spring+JPA CRUD（增删改查）也相当简单，让我们直接看具体示例吧：

```
@Test
public void testCRUD() {
    PlatformTransactionManager txManager = ctx.getBean(PlatformTransactionManager.class);
    final JpaTemplate jpaTemplate = new JpaTemplate(entityManagerFactory);
    TransactionTemplate tansactionTemplate = new TransactionTemplate(txManager);
    tansactionTemplate.execute(new TransactionCallback<Void>() {
        @Override
        public Void doInTransaction(TransactionStatus status) {
            UserModel model = new UserModel();
            model.setMyName("test");
            //新增
            jpaTemplate.persist(model);
            //修改
            model.setMyName("test2");
            jpaTemplate.flush();//可选
            //查询
            String sql = "from UserModel where myName=?";
            List result = jpaTemplate.find(sql, "test2");
            Assert.assertEquals(1, result.size());
            //删除
            jpaTemplate.remove(model);
            return null;
        }
    });
}
```

- 对于增删改必须运行在事务环境，因此我们使用TransactionTemplate事务模板类来支持事务。
- 持久化：使用JpaTemplate 类的persist方法持久化模型对象；
- 更新：对于持久化状态的模型对象直接修改属性，调用flush方法即可更新到数据库，在一些场合时flush方法调用可选，如执行一个查询操作等，具体请参考相关文档；
- 查询：可以使用find方法执行JPA QL查询；
- 删除：使用remove方法删除一个持久化状态的模型对象。

Spring集成JPA进行增删改查也相当简单，但本文介绍的稍微复杂一点，因为牵扯到编程式事务，如果采用声明式事务将和集成Hibernate方式一样简洁。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2500.html】

【第九章】 Spring的事务 之 9.1 数据库事务概述 ——跟我学spring3

9.1 数据库事务概述

事务首先是一系列操作组成的工作单元，该工作单元内的操作是不可分割的，即要么所有操作都做，要么所有操作都不做，这就是事务。

事务必需满足ACID（原子性、一致性、隔离性和持久性）特性，缺一不可：

- 原子性（**Atomicity**）：即事务是不可分割的最小工作单元，事务内的操作要么全做，要么全不做；
- 一致性（**Consistency**）：在事务执行前数据库的数据处于正确的状态，而事务执行完成后数据库的数据还是处于正确的状态，即数据完整性约束没有被破坏；如银行转帐，A转帐给B，必须保证A的钱一定转给B，一定不会出现A的钱转了但B没收到，否则数据库的数据就处于不一致（不正确）的状态。
- 隔离性（**Isolation**）：并发事务执行之间无影响，在一个事务内部的操作对其他事务是不产生影响，这需要事务隔离级别来指定隔离性；
- 持久性（**Durability**）：事务一旦执行成功，它对数据库的数据的改变必须是永久的，不会因比如遇到系统故障或断电造成数据不一致或丢失。

在实际项目开发中数据库操作一般都是并发执行的，即有多个事务并发执行，并发执行就可能遇到问题，目前常见的问题如下：

- 丢失更新：两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加锁造成的；
- 脏读：一个事务看到了另一个事务未提交的更新数据；
- 不可重复读：在同一事务中，多次读取同一数据却返回不同的结果；也就是有其他事务更改了这些数据；
- 幻读：一个事务在执行过程中读取到了另一个事务已提交的插入数据；即在第一个事务开始时读取到一批数据，但此后另一个事务又插入了新数据并提交，此时第一个事务又读取这批数据但发现多了一条，即好像发生幻觉一样。

为了解决这些并发问题，需要通过数据库隔离级别来解决，在标准SQL规范中定义了四种隔离级别：

- 未提交读（**Read Uncommitted**）：最低隔离级别，一个事务能读取到别的事务未提交的更新数据，很不安全，可能出现丢失更新、脏读、不可重复读、幻读；
- 提交读（**Read Committed**）：一个事务能读取到别的事务提交的更新数据，不能看到未提交的更新数据，不可能可能出现丢失更新、脏读，但可能出现不可重复读、幻读；

- 可重复读（**Repeatable Read**）：保证同一事务中先后执行的多次查询将返回同一结果，不受其他事务影响，可能可能出现丢失更新、脏读、不可重复读，但可能出现幻读；
- 序列化（**Serializable**）：最高隔离级别，不允许事务并发执行，而必须串行化执行，最安全，不可能出现更新、脏读、不可重复读、幻读。

隔离级别越高，数据库事务并发执行性能越差，能处理的操作越少。因此在实际项目开发中为了考虑并发性能一般使用提交读隔离级别，它能避免丢失更新和脏读，尽管不可重复读和幻读不能避免，但可以在可能出现的场合使用悲观锁或乐观锁来解决这些问题。

9.1.1 事务类型

数据库事务类型有本地事务和分布式事务：

- 本地事务：就是普通事务，能保证单台数据库上的操作的ACID，被限定在一台数据库上；
- 分布式事务：涉及两个或多个数据库源的事务，即跨越多台同类或异类数据库的事务（由每台数据库的本地事务组成的），分布式事务旨在保证这些本地事务的所有操作的ACID，使事务可以跨越多台数据库；

Java事务类型有JDBC事务和JTA事务：

- JDBC事务：就是数据库事务类型中的本地事务，通过Connection对象的控制来管理事务；
- JTA事务：JTA指Java事务API(Java Transaction API)，是Java EE数据库事务规范，JTA只提供了事务管理接口，由应用程序服务器厂商（如WebSphere Application Server）提供实现，JTA事务比JDBC更强大，支持分布式事务。

Java EE事务类型有本地事务和全局事务：

- 本地事务：使用JDBC编程实现事务；
- 全局事务：由应用程序服务器提供，使用JTA事务；

按是否通过编程实现事务有声明式事务和编程式事务；

- 声明式事务： 通过注解或XML配置文件指定事务信息；
- 编程式事务：通过编写代码实现事务。

9.1.2 Spring提供的事务管理

Spring框架最核心功能之一就是事务管理，而且提供一致的事务管理抽象，这能帮助我们：

- 提供一致的编程式事务管理API，不管使用Spring JDBC框架还是集成第三方框架使用该API进行事务编程；
- 无侵入式的声明式事务支持。

Spring支持声明式事务和编程式事务事务类型。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2502.html】

【第九章】 Spring的事务 之 9.2 事务管理器 ——跟我学spring3

9.2.1 概述

Spring框架支持事务管理的核心是事务管理器抽象，对于不同的数据访问框架（如Hibernate）通过实现策略接口PlatformTransactionManager，从而能支持各种数据访问框架的事务管理，PlatformTransactionManager接口定义如下：

```
public interface PlatformTransactionManager {
       TransactionStatus getTransaction(TransactionDefinition definition) throws Transact
       void commit(TransactionStatus status) throws TransactionException;
       void rollback(TransactionStatus status) throws TransactionException;
}
```

- **getTransaction()**：返回一个已经激活的事务或创建一个新的事务（根据给定的TransactionDefinition类型参数定义的事务属性），返回的是TransactionStatus对象代表了当前事务的状态，其中该方法抛出TransactionException（未检查异常）表示事务由于某种原因失败。
- **commit()**：用于提交TransactionStatus参数代表的事务，具体语义请参考Spring Javadoc；
- **rollback()**：用于回滚TransactionStatus参数代表的事务，具体语义请参考Spring Javadoc。

TransactionDefinition接口定义如下：

```
public interface TransactionDefinition {
       int getPropagationBehavior();
       int getIsolationLevel();
       int getTimeout();
       boolean isReadOnly();
       String getName();
}
```

- **getPropagationBehavior()**：返回定义的事务传播行为；
- **getIsolationLevel()**：返回定义的事务隔离级别；
- **getTimeout()**：返回定义的事务超时时间；
- **isReadOnly()**：返回定义的事务是否是只读的；
- **getName()**：返回定义的事务名字。

TransactionStatus接口定义如下：

```
public interface TransactionStatus extends SavepointManager {
       boolean isNewTransaction();
       boolean hasSavepoint();
       void setRollbackOnly();
       boolean isRollbackOnly();
       void flush();
       boolean isCompleted();
}
```

- **isNewTransaction()**：返回当前事务状态是否是新事务；
- **hasSavepoint()**：返回当前事务是否有保存点；
- **setRollbackOnly()**：设置当前事务应该回滚；
- **isRollbackOnly(()**：返回当前事务是否应该回滚；
- **flush()**：用于刷新底层会话中的修改到数据库，一般用于刷新如Hibernate/JPA的会话，可能对如JDBC类型的事务无任何影响；
- **isCompleted():**当前事务否已经完成。

9.2.2 内置事务管理器实现

Spring提供了许多内置事务管理器实现：

- **DataSourceTransactionManager**：位于org.springframework.jdbc.datasource包中，数据源事务管理器，提供对单个javax.sql.DataSource事务管理，用于Spring JDBC抽象框架、iBATIS或MyBatis框架的事务管理；
- **JdoTransactionManager**：位于org.springframework.orm.jdo包中，提供对单个javax.jdo.PersistenceManagerFactory事务管理，用于集成JDO框架时的事务管理；
- **JpaTransactionManager**：位于org.springframework.orm.jpa包中，提供对单个javax.persistence.EntityManagerFactory事务支持，用于集成JPA实现框架时的事务管理；
- **HibernateTransactionManager**：位于org.springframework.orm.hibernate3包中，提供对单个org.hibernate.SessionFactory事务支持，用于集成Hibernate框架时的事务管理；该事务管理器只支持Hibernate3+版本，且Spring3.0+版本只支持Hibernate 3.2+版本；
- **JtaTransactionManager**：位于org.springframework.transaction.jta包中，提供对分布式事务管理的支持，并将事务管理委托给Java EE应用服务器事务管理器；
- **OC4JjtaTransactionManager**：位于org.springframework.transaction.jta包中，Spring提供的对OC4J10.1.3+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持；
- **WebSphereUowTransactionManager**：位于org.springframework.transaction.jta包中，Spring提供的对WebSphere 6.0+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持；
- **WebLogicJtaTransactionManager**：位于org.springframework.transaction.jta包中，Spring提供的对WebLogic 8.1+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持。

Spring不仅提供这些事务管理器，还提供对如JMS事务管理的管理器等，Spring提供一致的事务抽象如图9-1所示。

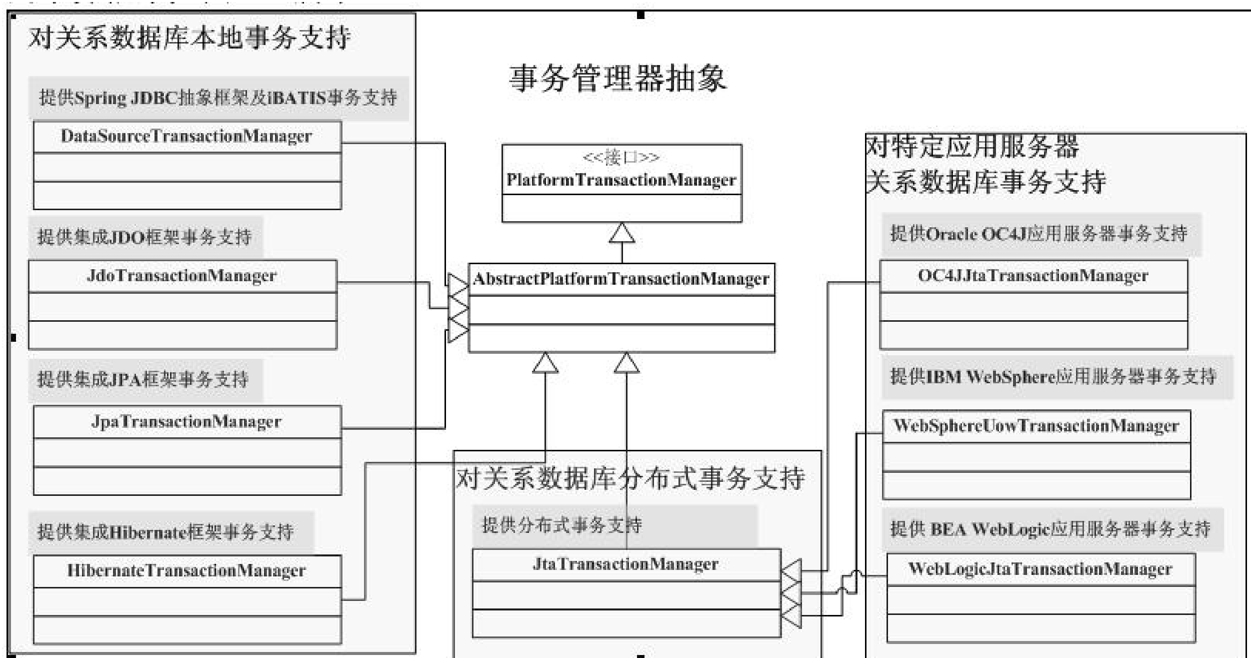


图9-1 Spring事务管理器

接下来让我们学习一下如何在Spring配置文件中定义事务管理器：

一、声明对本地事务的支持：

a)JDBC及iBATIS、MyBatis框架事务管理器

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionMana
    <property name="dataSource" ref="dataSource"/>
</bean>
```

通过dataSource属性指定需要事务管理的单个javax.sql.DataSource对象。

b)Jdo事务管理器

```
<bean id="txManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="persistenceManagerFactory"/>
</bean>
```

通过persistenceManagerFactory属性指定需要事务管理的javax.jdo.PersistenceManagerFactory对象。

c)Jpa事务管理器

```
<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

通过entityManagerFactory属性指定需要事务管理的javax.persistence.EntityManagerFactory对象。

还需要为entityManagerFactory对象指定jpaDialect属性，该属性所对应的对象指定了如何获取连接对象、开启事务、关闭事务等事务管理相关的行为。

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityMa
        ……
        <property name="jpaDialect" ref="jpaDialect"/>
</bean>
<bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
```

d)Hibernate事务管理器

```
<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManage
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

通过entityManagerFactory属性指定需要事务管理的org.hibernate.SessionFactory对象。

二、**Spring**对全局事务的支持：

a)Jta事务管理器

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/test"/>
  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName" value=" java:comp/TransactionManager"/>
  </bean>
</beans>
```

“dataSource”Bean表示从JNDI中获取的数据源，而txManager是JTA事务管理器，其中属性transactionManagerName指定了JTA事务管理器的JNDI名字，从而将事务管理委托给该事务管理器。

这只是最简单的配置方式，更复杂的形式请参考Spring Javadoc。

在此我们再介绍两个不依赖于应用服务器的开源JTA事务实现：JOTM和Atomikos Transactions Essentials。

- **JOTM**：即基于Java开放事务管理器（Java Open Transaction Manager），实现JTA规范，能够运行在非应用服务器环境中，Web容器或独立Java SE环境，官网地址:

http://jotm.objectweb.org/。

- **Atomikos Transactions Essentials**：其为Atomikos开发的事务管理器，该产品属于开源产品，另外还一个商业的Extreme Transactions。官网地址为：http://www.atomikos.com。

对于以上JTA事务管理器使用，本文作者只是做演示使用，如果在实际项目中需要不依赖于应用服务器的JTA事务支持，需详细测试并选择合适的。

在本文中将使用Atomikos Transactions Essentials来进行演示JTA事务使用，由于Atomikos对hsqldb分布式支持不是很好，在Atomikos官网中列出如下兼容的数据库：Oracle、Informix、FirstSQL、DB2、MySQL、SQLServer、Sybase，这不代表其他数据库不支持，而是Atomikos团队没完全测试，在此作者决定使用derby内存数据库来演示JTA分布式事务。

1、首先准备**jar**包：

1.1、准备**derby**数据**jar**包，到下载的spring-framework-3.0.5.RELEASE-dependencies.zip中拷贝如下jar包：

- com.springsource.org.apache.derby-10.5.1000001.764942.jar

1·**2**、准备**Atomikos Transactions Essentials** 对**JTA**事务支持的**JTA**包，此处使用AtomikosTransactionsEssentials3.5.5版本，到官网下载AtomikosTransactionsEssentials-3.5.5.zip，拷贝如下jar包到类路径：

- atomikos-util.jar
- transactions-api.jar
- transactions-essentials-all.jar
- transactions-jdbc.jar
- transactions-jta.jar
- transactions.jar

将如上jar包放在lib\atomikos目录下，并添加到类路径中。

2、接下来看一下在**Spring**中如何配置**AtomikosTransactionsEssentials JTA**事务：

2.1、配置分布式数据源：

```
<bean id="dataSource1" class="com.atomikos.jdbc.AtomikosDataSourceBean" init-method="init
    <property name="uniqueResourceName" value="jdbc/test1"/>
    <property name="xaDataSourceClassName" value="org.apache.derby.jdbc.EmbeddedXADataSou
    <property name="poolSize" value="5"/>
    <property name="xaProperties">
        <props>
            <prop key="databaseName">test1</prop>
            <prop key="createDatabase">create</prop>
        </props>
    </property>
</bean>

<bean id="dataSource2" class="com.atomikos.jdbc.AtomikosDataSourceBean"
      init-method="init" destroy-method="close">
    ……
</bean>
```

在此我们配置两个分布式数据源：使用com.atomikos.jdbc.AtomikosDataSourceBean来配置AtomikosTransactionsEssentials分布式数据源：

- uniqueResourceName表示唯一资源名，如有多个数据源不可重复；
- xaDataSourceClassName是具体分布式数据源厂商实现；
- poolSize是数据连接池大小；
- xaProperties属性指定具体厂商数据库属性，如databaseName指定数据库名，createDatabase表示启动derby内嵌数据库时创建databaseName指定的数据库。

在此我们还有定义了一个“dataSource2”Bean，其属性和“DataSource1”除以下不一样其他完全一样：

- uniqueResourceName：因为不能重复，因此此处使用jdbc/test2；
- databaseName：我们需要指定两个数据库，因此此处我们指定为test2。

2.2、配置事务管理器：

```
<bean id="atomikosTransactionManager" class = "com.atomikos.icatch.jta.UserTransactionMan
      <property name="forceShutdown" value="true"/>
</bean>
<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp">  <

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionMa
    <property name="transactionManager">
        <ref bean="atomikosTransactionManager"/>
    </property>
    <property name="userTransaction">
        <ref bean="atomikosUserTransaction"/>
    </property>
</bean>
```

- atomikosTransactionManager：定义了AtomikosTransactionsEssentials事务管理器；
- atomikosUserTransaction：定义UserTransaction，该Bean是线程安全的；
- transactionManager：定义Spring事务管理器，transactionManager属性指定外部事务管理器（真正的事务管理者），使用userTransaction指定UserTransaction，该属性一般用

于本地JTA实现，如果使用应用服务器事务管理器，该属性将自动从JNDI获取。

配置完毕，是不是也挺简单的，但是如果确实需要使用JTA事务，请首先选择应用服务器事务管理器，本示例不适合生产环境，如果非要运用到生产环境，可以考虑购买AtomikosTransactionsEssentials商业支持。

b)特定服务器事务管理器

Spring还提供了对特定应用服务器事务管理器集成的支持，目前提供对IBM WebSphere、BEA WebLogic、 Oracle OC4J应用服务器高级事务的支持，具体使用请参考Spring Javadoc。

现在我们已经学习如何配置事务管理器了，但是只有事务管理器Spring能自动进行事务管理吗？当然不能了，这需要我们来控制，目前Spring支持两种事务管理方式：编程式和声明式事务管理。接下来先看一下如何进行编程式事务管理吧。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2503.html】

【第九章】Spring的事务 之 9.3 编程式事务 ——跟我学spring3

9.3 编程式事务

9.3.1 编程式事务概述

所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。

Spring框架提供一致的事务抽象，因此对于JDBC还是JTA事务都是采用相同的API进行编程。

```
Connection conn = null;
UserTransaction tx = null;
try {
    tx = getUserTransaction();                       //1.获取事务
    tx.begin();                                    //2.开启JTA事务
    conn = getDataSource().getConnection();           //3.获取JDBC
    //4.声明SQL
    String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
    PreparedStatement pstmt = conn.prepareStatement(sql);//5.预编译SQL
    ResultSet rs = pstmt.executeQuery();               //6.执行SQL
    process(rs);                                   //7.处理结果集
    closeResultSet(rs);                             //8.释放结果集
    tx.commit();                                  //7.提交事务
} catch (Exception e) {
    tx.rollback();                                 //8.回滚事务
    throw e;
} finally {
   conn.close();                                //关闭连接
}
```

此处可以看到使用UserTransaction而不是Connection连接进行控制事务，从而对于JDBC事务和JTA事务是采用不同API进行编程控制的，并且JTA和JDBC事务管理的异常也是不一样的。

具体如何使用JTA编程进行事务管理请参考cn.javass.spring.chapter9包下的TranditionalTransactionTest类。

而在Spring中将采用一致的事务抽象进行控制和一致的异常控制，即面向PlatformTransactionManager接口编程来控制事务。

9.3.1 Spring对编程式事务的支持

Spring中的事务分为物理事务和逻辑事务；

- 物理事务：就是底层数据库提供的事务支持，如JDBC或JTA提供的事务；
- 逻辑事务：是Spring管理的事务，不同于物理事务，逻辑事务提供更丰富的控制，而且如果想得到Spring事务管理的好处，必须使用逻辑事务，因此在Spring中如果没特别强调一

般就是逻辑事务；

逻辑事务即支持非常低级别的控制，也有高级别解决方案：

- 低级别解决方案：

工具类：使用工具类获取连接（会话）和释放连接（会话），如使用 org.springframework.jdbc.datasource包中的 DataSourceUtils 类来获取和释放具有逻辑事务功能的连接。当然对集成第三方ORM框架也提供了类似的工具类，如对Hibernate提供了SessionFactoryUtils工具类，JPA的EntityManagerFactoryUtils等，其他工具类都是使用类似*Utils命名；

```
//获取具有Spring事务（逻辑事务）管理功能的连接
DataSourceUtils. getConnection(DataSource dataSource)
//释放具有Spring事务（逻辑事务）管理功能的连接
DataSourceUtils. releaseConnection(Connection con, DataSource dataSource)
```

TransactionAwareDataSourceProxy：使用该数据源代理类包装需要Spring事务管理支持的数据源，该包装类必须位于最外层，主要用于遗留项目中可能直接使用数据源获取连接和释放连接支持或希望在Spring中进行混合使用各种持久化框架时使用，其内部实际使用 DataSourceUtils 工具类获取和释放真正连接；

```
<!--使用该方式包装数据源，必须在最外层，targetDataSource 知道目标数据源-->
<bean id="dataSourceProxy"
class="org.springframework.jdbc.datasource.
TransactionAwareDataSourceProxy">
    <property name="targetDataSource" ref="dataSource"/>
</bean>
```

通过如上方式包装数据源后，可以在项目中使用物理事务编码的方式来获得逻辑事务的支持，即支持直接从DataSource获取连接和释放连接，且这些连接自动支持Spring逻辑事务；

- 高级别解决方案：

模板类：使用Spring提供的模板类，如JdbcTemplate、HibernateTemplate和JpaTemplate模板类等，而这些模板类内部其实是使用了低级别解决方案中的工具类来管理连接或会话；

Spring提供两种编程式事务支持：直接使用PlatformTransactionManager实现和使用TransactionTemplate模板类，用于支持逻辑事务管理。

如果采用编程式事务推荐使用TransactionTemplate模板类和高级别解决方案。

9.3.3 使用PlatformTransactionManager

首先让我们看下如何使用PlatformTransactionManager实现来进行事务管理：

1、数据源定义，此处使用第**7**章的配置文件，即**“chapter7/ applicationContext-resources.xml”**文件。

2、事务管理器定义（**chapter9/applicationContext-jdbc.xml**）：

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransa
    <property name="dataSource" ref="dataSource"/>
</bean>
```

3、 准备测试环境：

3.1、首先准备测试时使用的**SQL**：

```
package cn.javass.spring.chapter9;
//省略import
public class TransactionTest {
    //id自增主键从0开始
    private static final String CREATE_TABLE_SQL = "create table test" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
    private static final String DROP_TABLE_SQL = "drop table test";
    private static final String INSERT_SQL = "insert into test(name) values(?)";
    private static final String COUNT_SQL = "select count(*) from test";
    ……
}
```

3.2、初始化**Spring**容器

```
package cn.javass.spring.chapter9;
//省略import
public class TransactionTest {
    private static ApplicationContext ctx;
    private static PlatformTransactionManager txManager;
    private static DataSource dataSource;
    private static JdbcTemplate jdbcTemplate;
    ……
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter9/applicationContext-jdbc.xml"};
        ctx = new ClassPathXmlApplicationContext(configLocations);
        txManager = ctx.getBean(PlatformTransactionManager.class);
        dataSource = ctx.getBean(DataSource.class);
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    ……
}
```

3.3、使用高级别方案**JdbcTemplate**来进行事务管理器测试：

```
@Test
public void testPlatformTransactionManager() {
    DefaultTransactionDefinition def = new DefaultTransactionDefinition();
    def.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
    def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
    TransactionStatus status = txManager.getTransaction(def);
    jdbcTemplate.execute(CREATE_TABLE_SQL);
    try {
        jdbcTemplate.update(INSERT_SQL, "test");
        txManager.commit(status);
    } catch (RuntimeException e) {
        txManager.rollback(status);
    }
    jdbcTemplate.execute(DROP_TABLE_SQL);
}
```

- **DefaultTransactionDefinition**：事务定义，定义如隔离级别、传播行为等，即在本示例中隔离级别为ISOLATION_READ_COMMITTED（提交读），传播行为为PROPAGATION_REQUIRED（必须有事务支持，即如果当前没有事务，就新建一个事务，如果已经存在一个事务中，就加入到这个事务中）。
- **TransactionStatus**：事务状态类，通过PlatformTransactionManager的getTransaction方法根据事务定义获取；获取事务状态后，Spring根据传播行为来决定如何开启事务；
- JdbcTemplate：通过JdbcTemplate对象执行相应的SQL操作，且自动享受到事务支持，注意事务是线程绑定的，因此事务管理器可以运行在多线程环境；
- txManager.commit(status)：提交status对象绑定的事务；
- txManager.rollback(status)：当遇到异常时回滚status对象绑定的事务。

3.4、使用低级别解决方案来进行事务管理器测试：

```
@Test
public void testPlatformTransactionManagerForLowLevel1() {
DefaultTransactionDefinition def = new DefaultTransactionDefinition();      def.setIsolat
  TransactionStatus status = txManager.getTransaction(def);
  Connection conn = DataSourceUtils.getConnection(dataSource);
  try {
      conn.prepareStatement(CREATE_TABLE_SQL).execute();
      PreparedStatement pstmt = conn.prepareStatement(INSERT_SQL);
      pstmt.setString(1, "test");
      pstmt.execute();
      conn.prepareStatement(DROP_TABLE_SQL).execute();
      txManager.commit(status);
  } catch (Exception e) {
      status.setRollbackOnly();
      txManager.rollback(status);
  } finally {
      DataSourceUtils.releaseConnection(conn, dataSource);
}
}
```

低级别方案中使用DataSourceUtils获取和释放连接，使用txManager开管理事务，而且面向JDBC编程，比起模板类方式来繁琐和复杂的多，因此不推荐使用该方式。在此就不介绍数据源代理类使用了，需要请参考platformTransactionManagerForLowLevelTest2测试方法。

到此事务管理是不是还很繁琐？必须手工提交或回滚事务，有没有更好的解决方案呢？Spring提供了TransactionTemplate模板类来简化事务管理。

9.3.4 使用TransactionTemplate

TransactionTemplate模板类用于简化事务管理，事务管理由模板类定义，而具体操作需要通过TransactionCallback回调接口或TransactionCallbackWithoutResult回调接口指定，通过调用模板类的参数类型为TransactionCallback或TransactionCallbackWithoutResult的execute方法来自动享受事务管理。

TransactionTemplate模板类使用的回调接口：

- **TransactionCallback**：通过实现该接口的“T doInTransaction(TransactionStatus status) ”方法来定义需要事务管理的操作代码；
- **TransactionCallbackWithoutResult**：继承TransactionCallback接口，提供“void doInTransactionWithoutResult(TransactionStatus status)”便利接口用于方便那些不需要返回值的事务操作代码。

1、接下来演示一下**TransactionTemplate**模板类如何使用：

```
@Test
public void testTransactionTemplate() {//位于TransactionTest类中
  jdbcTemplate.execute(CREATE_TABLE_SQL);
  TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
  transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
  transactionTemplate.execute(new TransactionCallbackWithoutResult() {
      @Override
      protected void doInTransactionWithoutResult(TransactionStatus status) {
         jdbcTemplate.update(INSERT_SQL, "test");
  }});
  jdbcTemplate.execute(DROP_TABLE_SQL);
}
```

- **TransactionTemplate** ：通过new TransactionTemplate(txManager)创建事务模板类，其中构造器参数为PlatformTransactionManager实现，并通过其相应方法设置事务定义，如事务隔离级别、传播行为等，此处未指定传播行为，其默认为PROPAGATION_REQUIRED；
- TransactionCallbackWithoutResult：此处使用不带返回的回调实现，其doInTransactionWithoutResult方法实现中定义了需要事务管理的操作；
- transactionTemplate.execute()：通过该方法执行需要事务管理的回调。

这样是不是简单多了，没有事务管理代码，而是由模板类来完成事务管理。

注：对于抛出**Exception**类型的异常且需要回滚时，需要捕获异常并通过调用**status**对象的**setRollbackOnly()**方法告知事务管理器当前事务需要回滚，如下所示：

```
try {
    //业务操作
} catch (Exception e) { //可使用具体业务异常代替
    status.setRollbackOnly();
}
```

2、前边已经演示了**JDBC**事务管理，接下来演示一下**JTA**分布式事务管理：

```
@Test
public void testJtaTransactionTemplate() {
    String[] configLocations = new String[] {
      "classpath:chapter9/applicationContext-jta-derby.xml"};
    ctx = new ClassPathXmlApplicationContext(configLocations);
    final PlatformTransactionManager jtaTXManager = ctx.getBean(PlatformTransactionManage
    final DataSource derbyDataSource1 = ctx.getBean("dataSource1", DataSource.class);
    final DataSource derbyDataSource2 = ctx.getBean("dataSource2", DataSource.class);
    final JdbcTemplate jdbcTemplate1 = new JdbcTemplate(derbyDataSource1);
    final JdbcTemplate jdbcTemplate2 = new JdbcTemplate(derbyDataSource2);
    TransactionTemplate transactionTemplate = new TransactionTemplate(jtaTXManager);
    transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED)
    jdbcTemplate1.update(CREATE_TABLE_SQL);
    int originalCount = jdbcTemplate1.queryForInt(COUNT_SQL);
    try {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                jdbcTemplate1.update(INSERT_SQL, "test");
               //因为数据库2没有创建数据库表因此会回滚事务
              jdbcTemplate2.update(INSERT_SQL, "test");
          }});
    } catch (RuntimeException e) {
        int count = jdbcTemplate1.queryForInt(COUNT_SQL);
        Assert.assertEquals(originalCount, count);
    }
    jdbcTemplate1.update(DROP_TABLE_SQL);
}
```

- 配置文件：使用此前定义的chapter9/applicationContext-jta-derby.xml；
- **jtaTXManager**： JTA事务管理器；
- **derbyDataSource1**和**derbyDataSource2**：derby数据源1和derby数据源2；
- **jdbcTemplate1**和**jdbcTemplate2**：分别使用derbyDataSource1和derbyDataSource2构造的JDBC模板类；
- **transactionTemplate**：使用jtaTXManager事务管理器的事务管理模板类，其隔离级别为提交读，传播行为默认为PROPAGATION_REQUIRED（必须有事务支持，即如果当前没有事务，就新建一个事务，如果已经存在一个事务中，就加入到这个事务中）；
- **jdbcTemplate1.update(CREATE_TABLE_SQL)**：此处只有derbyDataSource1所代表的数据库创建了“test”表，而derbyDataSource2所代表的数据库没有此表；
- **TransactionCallbackWithoutResult**：在此接口实现中定义了需要事务支持的操作：

jdbcTemplate1.update(INSERT_SQL, "test")：表示向数据库1中的test表中插入数据；

jdbcTemplate2.update(INSERT_SQL, "test")：表示向数据库2中的test表中插入数据，但数据库2没有此表将抛出异常，且JTA分布式事务将回滚；

- **Assert.assertEquals(originalCount, count)**：用来验证事务是否回滚，验证结果返回为true，说明分布式事务回滚了。

到此我们已经会使用PlatformTransactionManager和TransactionTemplate进行简单事务处理了，那如何应用到实际项目中去呢？接下来让我们看下如何在实际项目中应用Spring管理事务。

接下来看一下如何将Spring管理事务应用到实际项目中，为简化演示，此处只定义最简单的模型对象和不完整的Dao层接口和Service层接口：

1、 首先定义项目中的模型对象，本示例使用用户模型和用户地址模型：

模型对象一般放在项目中的**model**包里。

```
package cn.javass.spring.chapter9.model;
public class UserModel {
    private int id;
    private String name;
    private AddressModel address;
    //省略getter和setter
}
```

```
package cn.javass.spring.chapter9.model;
public class AddressModel {
    private int id;
    private String province;
    private String city;
    privateString street;
    private int userId;
    //省略getter和setter
}
```

2.1、定义**Dao**层接口：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.UserModel;
public interface IUserService {
    public void save(UserModel user);
    public int countAll();
}
```

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.AddressModel;
public interface IAddressService {
    public void save(AddressModel address);
    public int countAll();
}
```

2.2、定义Dao层实现：

```
package cn.javass.spring.chapter9.dao.jdbc;
//省略import，注意model要引用chapter包里的
public class UserJdbcDaoImpl extends NamedParameterJdbcDaoSupport implements IUserDao {
    private final String INSERT_SQL = "insert into user(name) values(:name)";
    private final String COUNT_ALL_SQL = "select count(*) from user";
    @Override
    public void save(UserModel user) {
        KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
        SqlParameterSource paramSource = new BeanPropertySqlParameterSource(user);
        getNamedParameterJdbcTemplate().update(INSERT_SQL, paramSource, generatedKeyHolde
        user.setId(generatedKeyHolder.getKey().intValue());
    }
    @Override
    public int countAll() {
       return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
    }
}
```

```
package cn.javass.spring.chapter9.dao.jdbc;
//省略import，注意model要引用chapter包里的
public class AddressJdbcDaoImpl extends NamedParameterJdbcDaoSupport implements IAddressD
    private final String INSERT_SQL = "insert into address(province, city, street, user_i
    private final String COUNT_ALL_SQL = "select count(*) from address";
    @Override
    public void save(AddressModel address) {
        KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
        SqlParameterSource paramSource = new BeanPropertySqlParameterSource(address);
        getNamedParameterJdbcTemplate().update(INSERT_SQL, paramSource, generatedKeyHolde
        address.setId(generatedKeyHolder.getKey().intValue());
}
    @Override
    public int countAll() {
        return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
    }
}
```

3.1、定义Service层接口，一般使用“I×××Service”命名：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.UserModel;
public interface IUserService {
    public void save(UserModel user);
    public int countAll();
}

package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.AddressModel;
public interface IAddressService {
    public void save(AddressModel address);
    public int countAll();
}
```

3.2、定义Service层实现，一般使用“×××ServiceImpl”或“×××Service”命名：

```
package cn.javass.spring.chapter9.service.impl;
//省略import，注意model要引用chapter包里的
public class AddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    private PlatformTransactionManager txManager;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
    public void setTxManager(PlatformTransactionManager txManager) {
        this.txManager = txManager;
    }
    @Override
    public void save(final AddressModel address) {
        TransactionTemplate transactionTemplate = TransactionTemplateUtils.getDefaultTran
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
           @Override
           protected void doInTransactionWithoutResult(TransactionStatus status) {
                addressDao.save(address);
           }
        });
    }
    @Override
    public int countAll() {
        return addressDao.countAll();
    }
}
```

```
package cn.javass.spring.chapter9.service.impl;
//省略import，注意model要引用chapter包里的
public class UserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    private PlatformTransactionManager txManager;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setTxManager(PlatformTransactionManager txManager) {
        this.txManager = txManager;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Override
    public void save(final UserModel user) {
        TransactionTemplate transactionTemplate =
            TransactionTemplateUtils.getDefaultTransactionTemplate(txManager);
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
        @Override
           protected void doInTransactionWithoutResult(TransactionStatus status) {
                userDao.save(user);
                user.getAddress().setUserId(user.getId());
                addressService.save(user.getAddress());
           }
        });
    }
    @Override
    public int countAll() {
        return userDao.countAll();
    }
}
```

Service实现中需要Spring事务管理的部分应该使用TransactionTemplate模板类来包装执行。

4、定义**TransactionTemplateUtils**，用于简化获取**TransactionTemplate**模板类，工具类一般放在**util**包中：

```
package cn.javass.spring.chapter9.util;
//省略import
public class TransactionTemplateUtils {
    public static TransactionTemplate getTransactionTemplate(
            PlatformTransactionManager txManager,
            int propagationBehavior,
            int isolationLevel) {

        TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
        transactionTemplate.setPropagationBehavior(propagationBehavior);
        transactionTemplate.setIsolationLevel(isolationLevel);
        return transactionTemplate;
    }

    public static TransactionTemplate getDefaultTransactionTemplate(PlatformTransactionMa
        return getTransactionTemplate(
                txManager,
                TransactionDefinition.PROPAGATION_REQUIRED,
                TransactionDefinition.ISOLATION_READ_COMMITTED);
    }
}
```

getDefaultTransactionTemplate用于获取传播行为为PROPAGATION_REQUIRED，隔离级别为ISOLATION_READ_COMMITTED的模板类。

5、数据源配置定义，此处使用第**7**章的配置文件，即**"chapter7/ applicationContext-resources.xml"**文件。

6、**Dao**层配置定义（**chapter9/dao/applicationContext-jdbc.xml**）：

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionMana
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="abstractDao" abstract="true">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
<bean id="userDao" class="cn.javass.spring.chapter9.dao.jdbc.UserJdbcDaoImpl" parent="abs
<bean id="addressDao" class="cn.javass.spring.chapter9.dao.jdbc.AddressJdbcDaoImpl" paren
```

7、**Service**层配置定义（**chapter9/service/applicationContext-service.xml**）：

```
<bean id="userService" class="cn.javass.spring.chapter9.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="txManager" ref="txManager"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService" class="cn.javass.spring.chapter9.service.impl.AddressServiceImp
    <property name="addressDao" ref="addressDao"/>
    <property name="txManager" ref="txManager"/>
</bean>
```

8、准备测试需要的表创建语句，在**TransactionTest**测试类中添加如下静态变量：

```
private static final String CREATE_USER_TABLE_SQL =
    "create table user" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
private static final String DROP_USER_TABLE_SQL = "drop table user";

private static final String CREATE_ADDRESS_TABLE_SQL =
    "create table address" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "province varchar(100), city varchar(100), street varchar(100), user_id int)";
private static final String DROP_ADDRESS_TABLE_SQL = "drop table address";
```

9、 测试一下吧：

```
@Test
    public void testServiceTransaction() {
        String[] configLocations = new String[] {
        "classpath:chapter7/applicationContext-resources.xml",
        "classpath:chapter9/dao/applicationContext-jdbc.xml",
        "classpath:chapter9/service/applicationContext-service.xml"};
        ApplicationContext ctx2 = new ClassPathXmlApplicationContext(configLocations);

        DataSource dataSource2 = ctx2.getBean(DataSource.class);
        JdbcTemplate jdbcTemplate2 = new JdbcTemplate(dataSource2);
        jdbcTemplate2.update(CREATE_USER_TABLE_SQL);
        jdbcTemplate2.update(CREATE_ADDRESS_TABLE_SQL);

        IUserService userService = ctx2.getBean("userService", IUserService.class);
        IAddressService addressService = ctx2.getBean("addressService", IAddressService.c
        UserModel user = createDefaultUserModel();
        userService.save(user);
        Assert.assertEquals(1, userService.countAll());
        Assert.assertEquals(1, addressService.countAll());
        jdbcTemplate2.update(DROP_USER_TABLE_SQL);
        jdbcTemplate2.update(DROP_ADDRESS_TABLE_SQL);
}
private UserModel createDefaultUserModel() {
    UserModel user = new UserModel();
    user.setName("test");
    AddressModel address = new AddressModel();
    address.setProvince("beijing");
    address.setCity("beijing");
    address.setStreet("haidian");
    user.setAddress(address);
    return user;
}
```

从Spring容器中获取Service层对象，调用Service层对象持久化对象，大家有没有注意到Spring事务全部在Service层定义，为什么会在Service层定义，而不是Dao层定义呢？这是因为在服务层可能牵扯到业务逻辑，而每个业务逻辑可能调用多个Dao层方法，为保证这些操作的原子性，必须在Service层定义事务。

还有大家有没有注意到如果Service层的事务管理相当令人头疼，而且是侵入式的，有没有办法消除这些冗长的事务管理代码呢？这就需要Spring声明式事务支持，下一节将介绍无侵入式的声明式事务。

可能大家对事务定义中的各种属性有点困惑，如传播行为到底干什么用的？接下来将详细讲解一下事务属性。

9.3.5 事务属性

事务属性通过TransactionDefinition接口实现定义，主要有事务隔离级别、事务传播行为、事务超时时间、事务是否只读。

Spring提供TransactionDefinition接口默认实现DefaultTransactionDefinition，可以通过该实现类指定这些事务属性。

- 事务隔离级别：用来解决并发事务时出现的问题，其使用TransactionDefinition中的静态变量来指定：

ISOLATION_DEFAULT：默认隔离级别，即使用底层数据库默认的隔离级别；

ISOLATION_READ_UNCOMMITTED：未提交读；

ISOLATION_READ_COMMITTED：提交读，一般情况下我们使用这个；

ISOLATION_REPEATABLE_READ：可重复读；

ISOLATION_SERIALIZABLE：序列化。

可以使用DefaultTransactionDefinition类的setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED)来指定隔离级别，其中此处表示隔离级别为提交读，也可以使用或setIsolationLevelName(“ISOLATION_READ_COMMITTED”)方式指定，其中参数就是隔离级别静态变量的名字，但不推荐这种方式。

- 事务传播行为：Spring管理的事务是逻辑事务，而且物理事务和逻辑事务最大差别就在于事务传播行为，事务传播行为用于指定在多个事务方法间调用时，事务是如何在这些方法间传播的，Spring共支持7种传播行为：

Required：必须有逻辑事务，否则新建一个事务，使用PROPAGATION_REQUIRED指定，表示如果当前存在一个逻辑事务，则加入该逻辑事务，否则将新建一个逻辑事务，如图9-2和9-3所示；

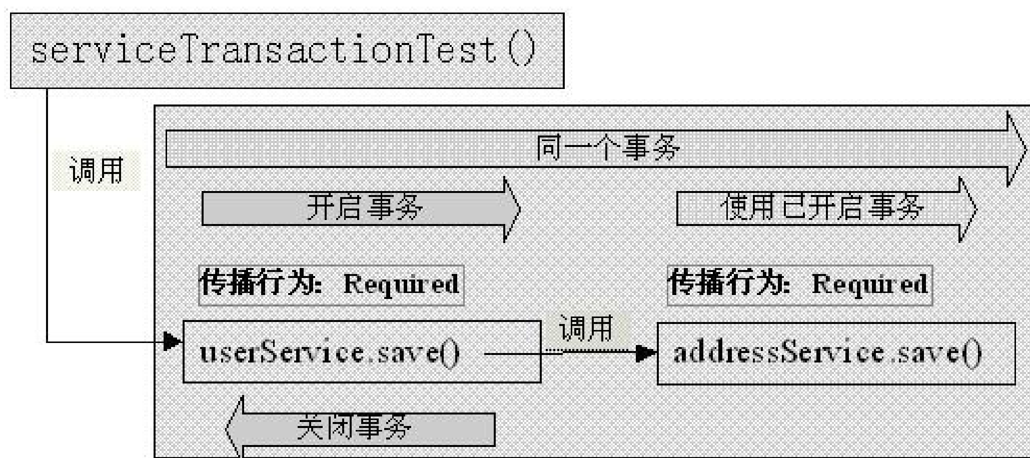


图9-2 Required传播行为

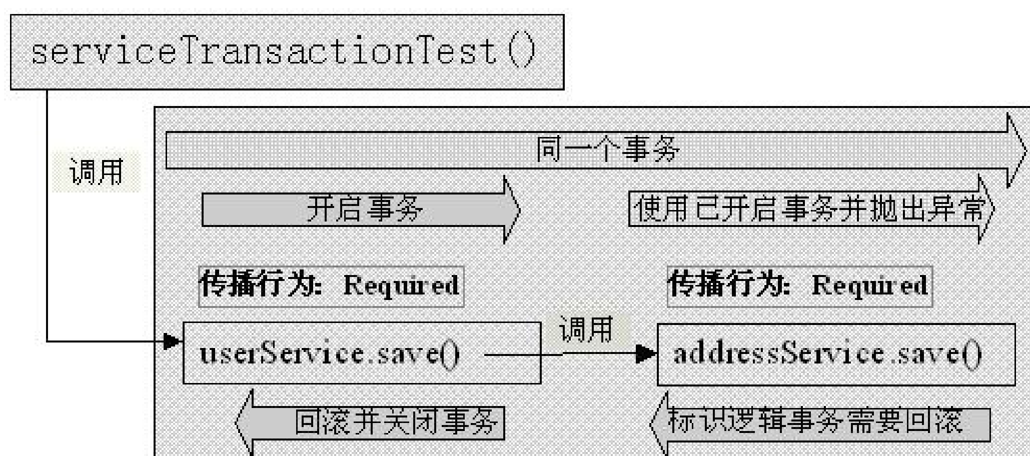


图9-3 Required传播行为抛出异常情况

在前边示例中就是使用的Required传播行为：

一、在调用userService对象的save方法时，此方法用的是Required传播行为且此时Spring事务管理器发现还没开启逻辑事务，因此Spring管理器觉得开启逻辑事务，

二、在此逻辑事务中调用了addressService对象的save方法，而在save方法中发现同样用的是Required传播行为，因此使用该已经存在的逻辑事务；

三、在返回到addressService对象的save方法，当事务模板类执行完毕，此时提交并关闭事务。

因此userService对象的save方法和addressService的save方法属于同一个物理事务，如果发生回滚，则两者都回滚。

接下来测试一下该传播行为如何执行吧：

一、正确提交测试，如上一节的测试，在此不再演示；

二、回滚测试，修改AddressServiceImpl的save方法片段：

```
addressDao.save(address);
```

为

```
addressDao.save(address);
//抛出异常，将标识当前事务需要回滚
throw new RuntimeException();
```

二、修改UserServiceImpl的save方法片段：

```
addressService.save(user.getAddress());
```

为

```
try {
    addressService.save(user.getAddress());//将在同一个事务内执行
} catch (RuntimeException e) {
}
```

如果该业务方法执行时事务被标记为回滚，则不管在此是否捕获该异常都将发生回滚，因为处于同一逻辑事务。

三、修改测试方法片段:

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

为如下形式：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(0, addressService.countAll());
```

Assert断言中countAll方法都返回0，说明事务回滚了，即说明两个业务方法属于同一个物理事务，即使在userService对象的save方法中将异常捕获，由于addressService对象的save方法抛出异常，即事务管理器将自动标识当前事务为需要回滚。

RequiresNew：创建新的逻辑事务，使用PROPAGATION_REQUIRES_NEW指定，表示每次都创建新的逻辑事务（物理事务也是不同的）如图9-4和9-5所示：

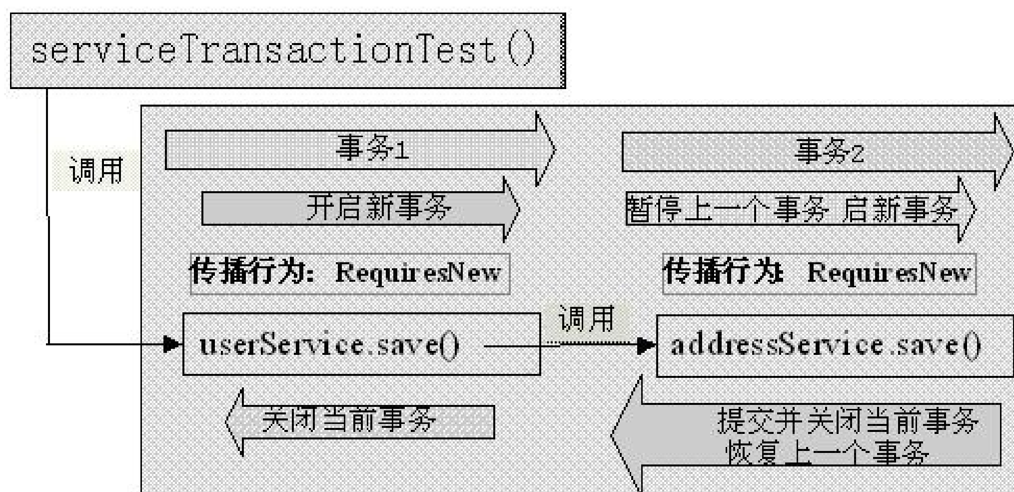


图9-4 RequiresNew传播行为

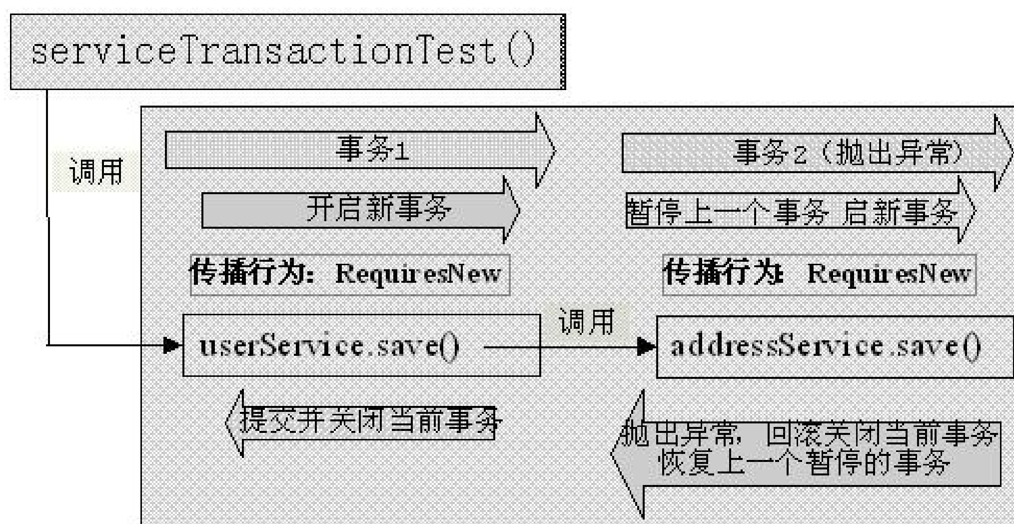


图9-5 RequiresNew传播行为并抛出异常

接下来测试一个该传播行为如何执行吧：

1、将如下获取事务模板方式

```
TransactionTemplate transactionTemplate = TransactionTemplateUtils.getDefaultTransactionT
```

替换为如下形式，表示传播行为为RequiresNew：

```
TransactionTemplate transactionTemplate = TransactionTemplateUtils.getTransactionTemplate
        txManager,
        TransactionDefinition.PROPAGATION_REQUIRES_NEW,
        TransactionDefinition.ISOLATION_READ_COMMITTED);
```

2、执行如下测试，发现执行结果是正确的：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

3、修改UserServiceImpl的save方法片段

```
userDao.save(user);
user.getAddress().setUserId(user.getId());
addressService.save(user.getAddress());
```

为如下形式，表示userServiceImpl类的save方法将发生回滚，而AddressServiceImpl类的方法由于在抛出异常前执行，将成功提交事务到数据库：

```
userDao.save(user);
user.getAddress().setUserId(user.getId());
addressService.save(user.getAddress());
throw new RuntimeException();
```

4、修改测试方法片段：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

为如下形式：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

Assert断言中调用userService对象countAll方法返回0，说明该逻辑事务作用域回滚，而调用addressService对象的countAll方法返回1，说明该逻辑事务作用域正确提交。因此这是不正确的行为，因为用户和地址应该是一一对应的，不应该发生这种情况，因此此处正确的传播行为应该是Required。

该传播行为执行流程（正确提交情况）：

一、当执行userService对象的save方法时，由于传播行为是RequiresNew，因此创建一个新的逻辑事务（物理事务也是不同的）；

二、当执行到addressService对象的save方法时，由于传播行为是RequiresNew，因此首先暂停上一个逻辑事务并创建一个新的逻辑事务（物理事务也是不同的）；

三、addressService对象的save方法执行完毕后，提交逻辑事务（并提交物理事务）并重新恢复上一个逻辑事务，继续执行userService对象的save方法内的操作；

四、最后userService对象的save方法执行完毕，提交逻辑事务（并提交物理事务）；

五、userService对象的save方法和addressService对象的save方法不属于同一个逻辑事务且也不属于同一个物理事务。

Supports：支持当前事务，使用PROPAGATION_SUPPORTS指定，指如果当前存在逻辑事务，就加入到该逻辑事务，如果当前没有逻辑事务，就以非事务方式执行，如图9-6和9-7所示：

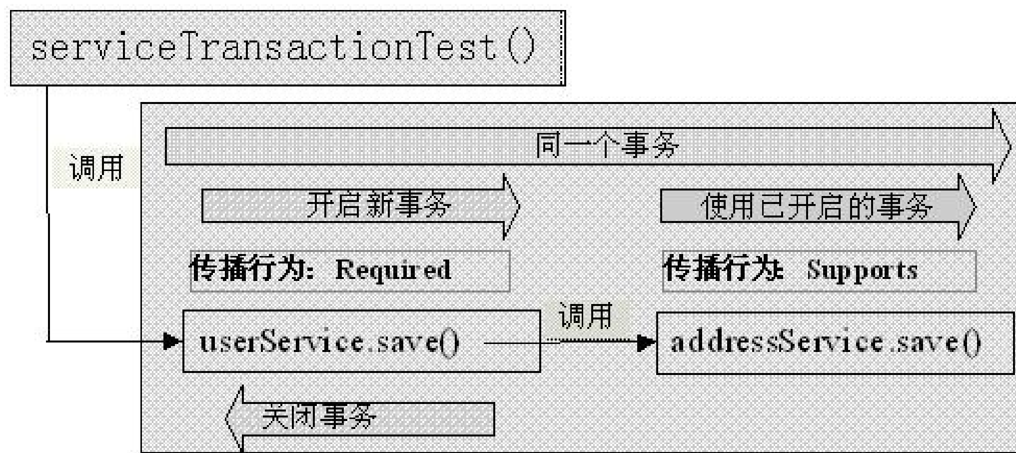


图9-6 Required+Supports传播行为

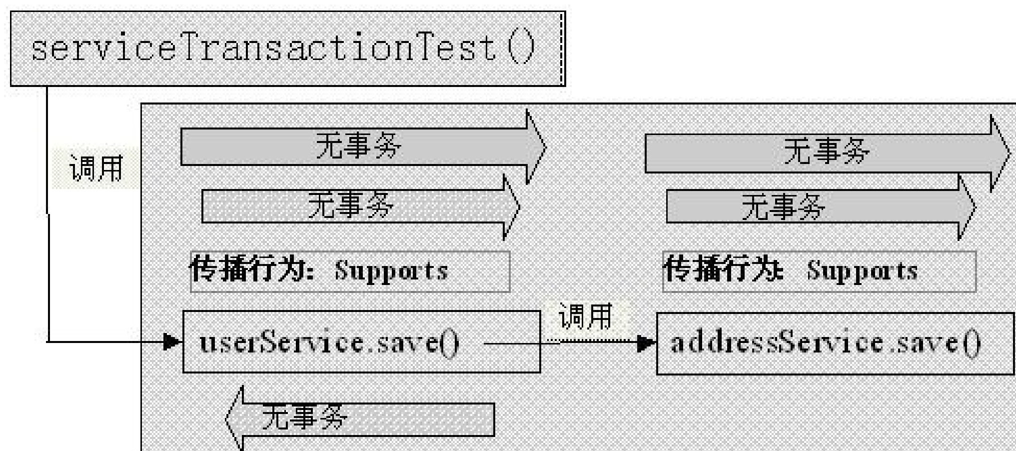


图9-7 Supports+Supports传播行为

NotSupported：不支持事务，如果当前存在事务则暂停该事务，使用PROPAGATION_NOT_SUPPORTED指定，即以非事务方式执行，如果当前存在逻辑事务，就把当前事务暂停，以非事务方式执行，如图9-8和9-9所示：

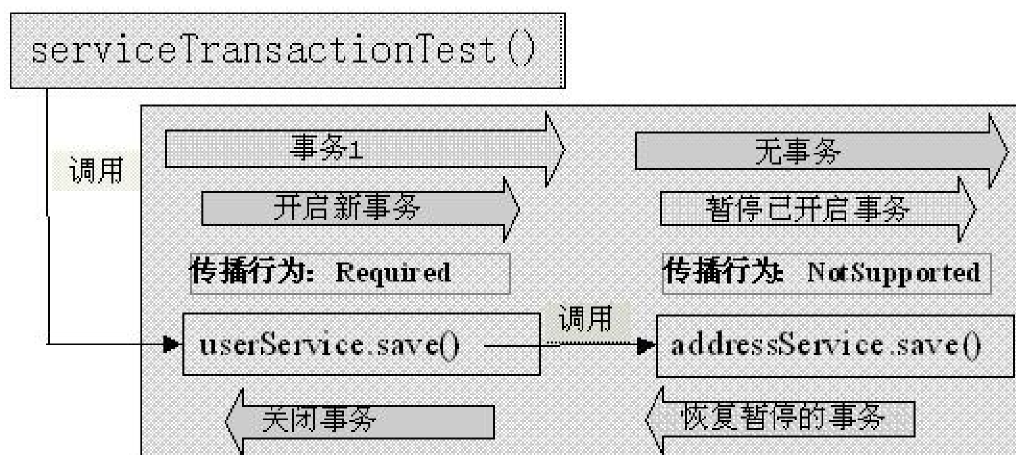


图9-8 Required+NotSupported传播行为

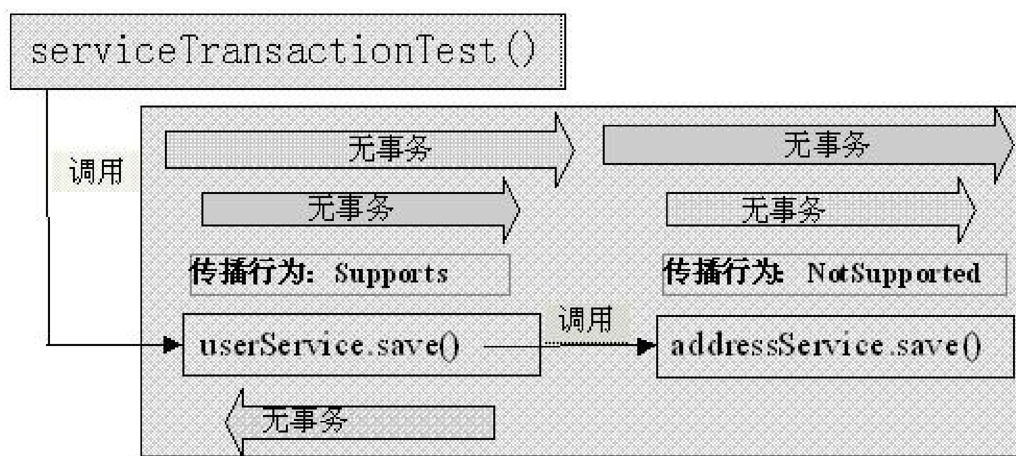


图9-9 Supports+NotSupported传播行为

Mandatory：必须有事务，否则抛出异常，使用PROPAGATION_MANDATORY指定，使用当前事务执行，如果当前没有事务，则抛出异常（IllegalTransactionStateException），如图9-10和9-11所示：

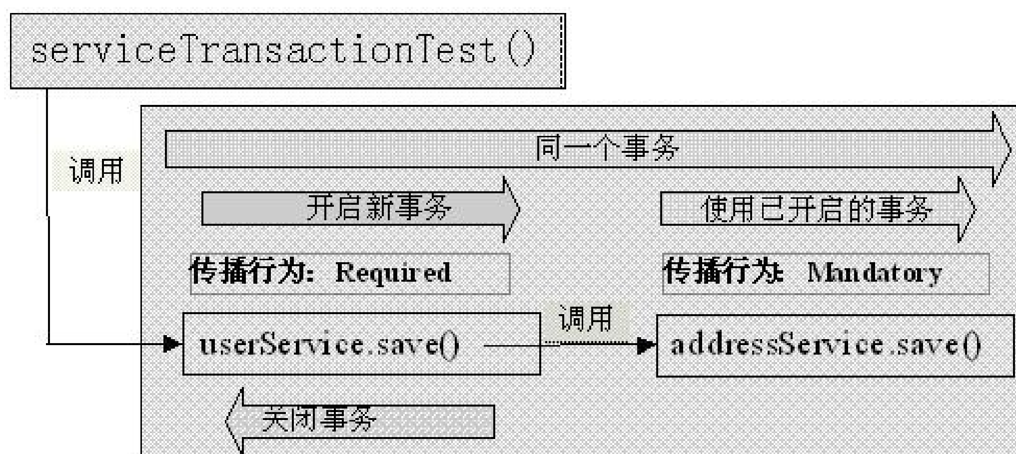


图9-10 Required+Mandatory传播行为

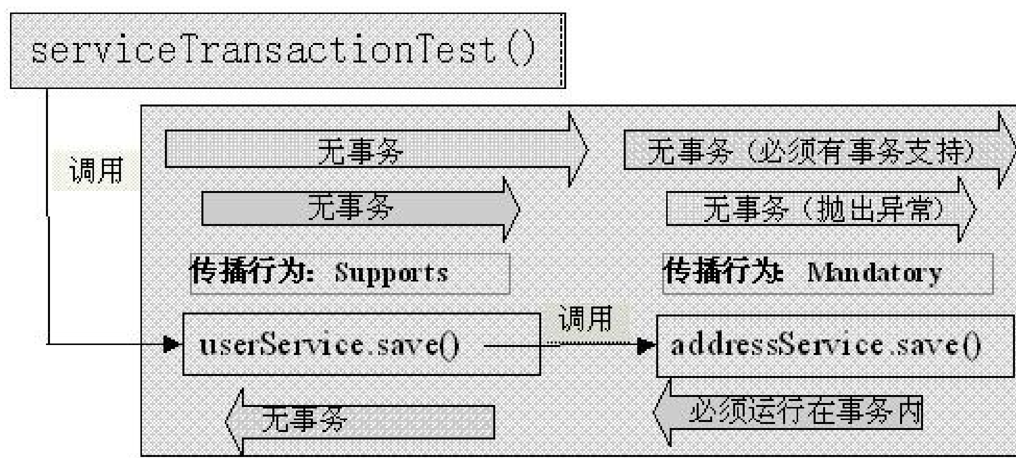


图9-11 Supports+Mandatory传播行为

Never：不支持事务，如果当前存在是事务则抛出异常，使用PROPAGATION_NEVER指定，即以非事务方式执行，如果当前存在事务，则抛出异常（IllegalTransactionStateException），如图9-12和9-13所示：

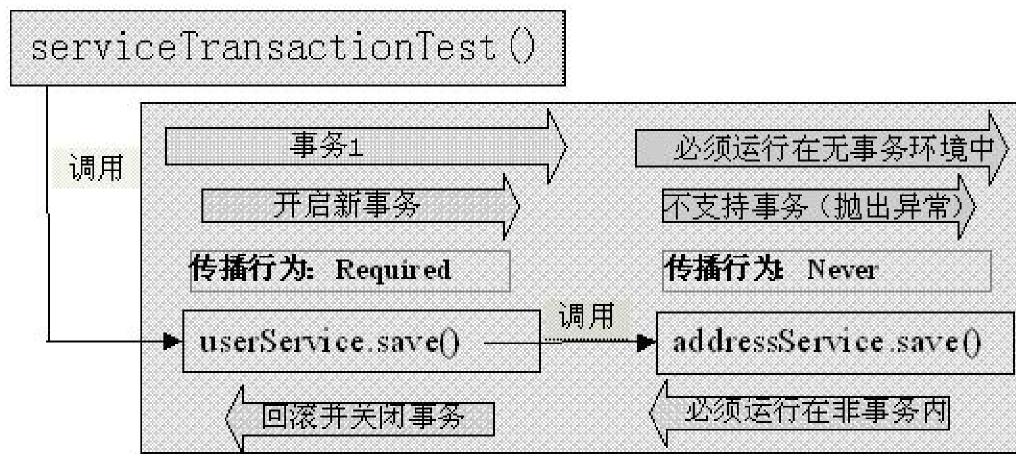


图9-12 Required+Never传播行为

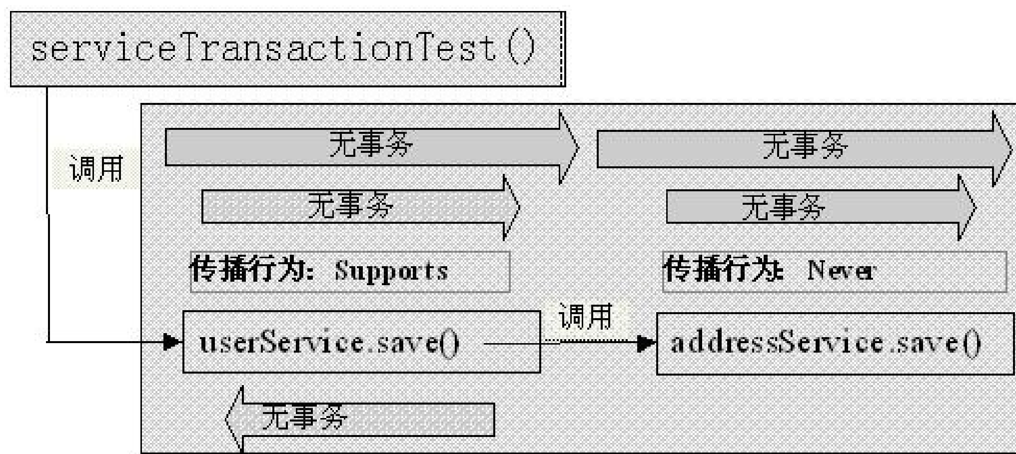


图9-13 Supports+Never传播行为

Nested：嵌套事务支持，使用PROPAGATION_NESTED指定，如果当前存在事务，则在嵌套事务内执行，如果当前不存在事务，则创建一个新的事务，嵌套事务使用数据库中的保存点来实现，即嵌套事务回滚不影响外部事务，但外部事务回滚将导致嵌套事务回滚，如图9-14和9-15所示：

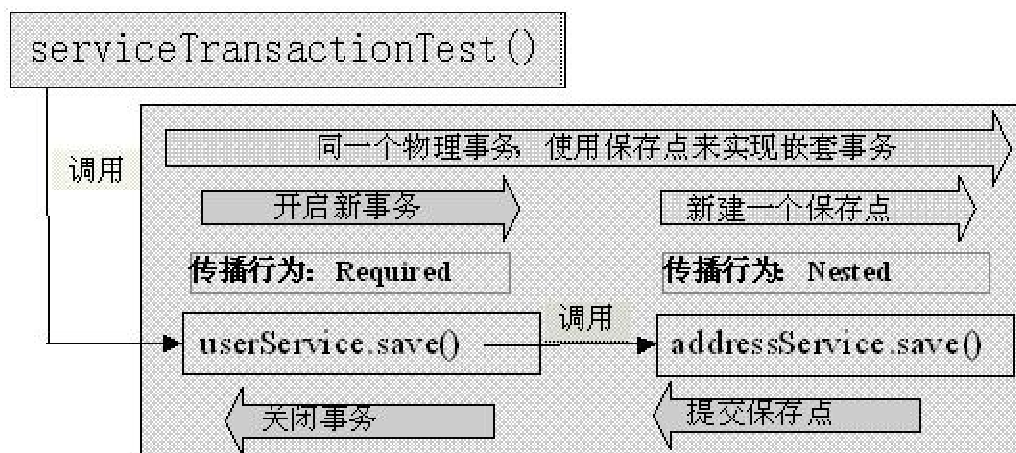


图9-14 Required+Nested传播行为

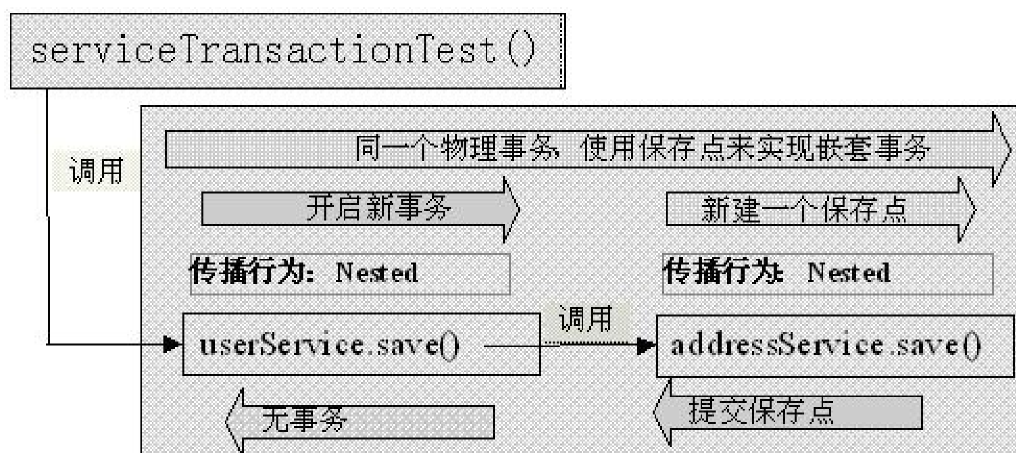


图9-15 Nested+Nested传播行为

Nested和**RequiresNew**的区别：

1、 RequiresNew每次都创建新的独立的物理事务，而Nested只有一个物理事务；

2、 Nested嵌套事务回滚或提交不会导致外部事务回滚或提交，但外部事务回滚将导致嵌套事务回滚，而 RequiresNew由于都是全新的事务，所以之间是无关联的；

3、 Nested使用JDBC 3的保存点实现，即如果使用低版本驱动将导致不支持嵌套事务。

使用嵌套事务，必须确保具体事务管理器实现的nestedTransactionAllowed属性为true，否则不支持嵌套事务，如DataSourceTransactionManager默认支持，而HibernateTransactionManager默认不支持，需要我们来开启。

对于事务传播行为我们只演示了Required和RequiresNew，其他传播行为类似，如果对这些事务传播行为不太会使用，请参考chapter9包下的TransactionTest测试类中的testPropagation方法，方法内有详细示例。

- 事务超时：设置事务的超时时间，单位为秒，默认为-1表示使用底层事务的超时时间；

使用如setTimeout(100)来设置超时时间，如果事务超时将抛出org.springframework.transaction.TransactionTimedOutException异常并将当前事务标记为应该回滚，即超时后事务被自动回滚；

可以使用具体事务管理器实现的defaultTimeout属性设置默认的事务超时时间，如DataSourceTransactionManager. setDefaultTimeout(10)。

- 事务只读：将事务标识为只读，只读事务不修改任何数据；

对于JDBC只是简单的将连接设置为只读模式，对于更新将抛出异常；

而对于一些其他ORM框架有一些优化作用，如在Hibernate中，Spring事务管理器将执行"session.setFlushMode(FlushMode.MANUAL)"即指定Hibernate会话在只读事务模式下不用尝试检测和同步持久对象的状态的更新。

如果使用设置具体事务管理的validateExistingTransaction属性为true（默认false），将确保整个事务传播链都是只读或都不是只读，如图9-16是正确的事务只读设置，而图9-17是错误的事务只读设置：

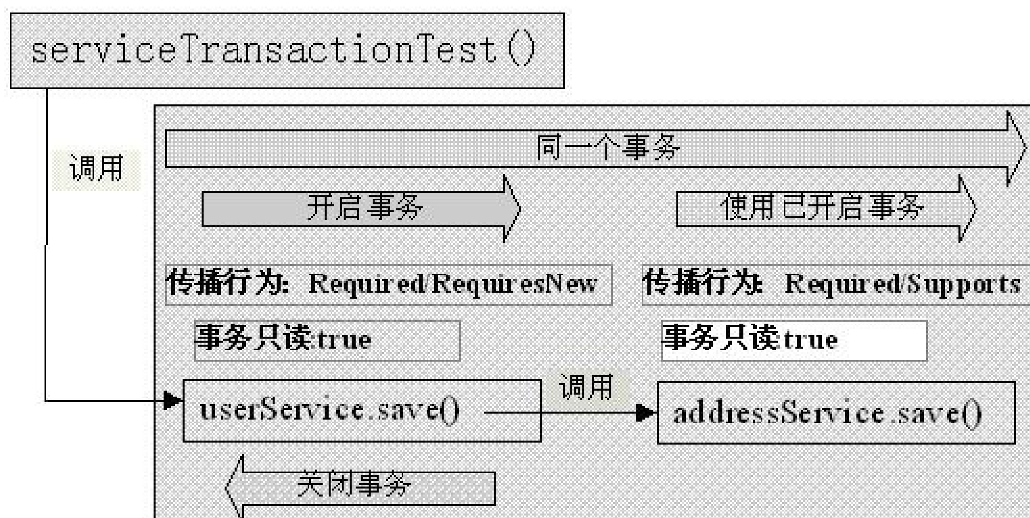


图9-16 正确的事务只读设置

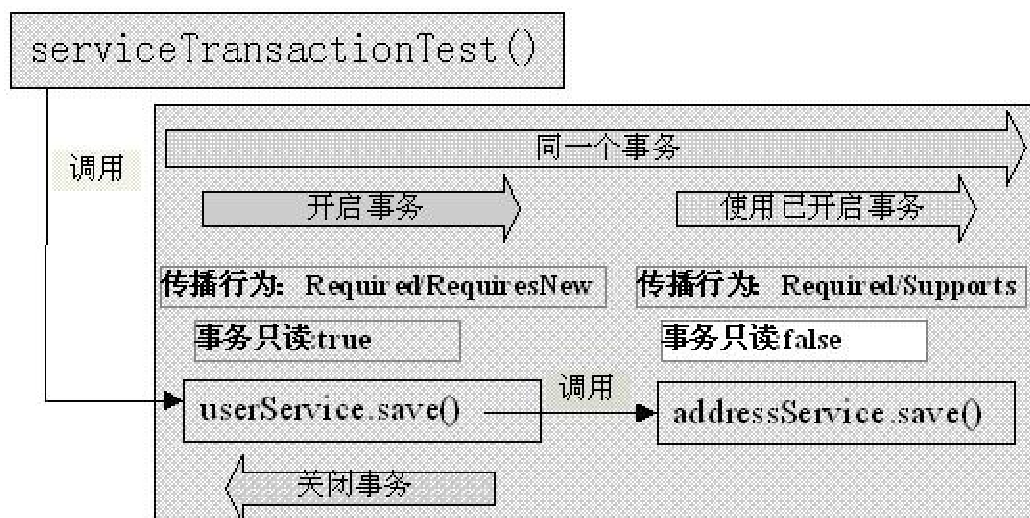


图9-17 错误的事务只读设置

如图10-17，对于错误的事务只读设置将抛出IllegalTransactionStateException异常，并伴随"Participating transaction with definition [……] is not marked as read-only……"信息，表示参与的事务只读属性设置错误。

大家有没有感觉到编程式实现事务管理是不是很繁琐冗长，重复，而且是侵入式的，因此发展到这Spring决定使用配置方式实现事务管理。

9.3.6 配置方式实现事务管理

在Spring2.x之前为了解决编程式事务管理的各种不好问题，Spring提出使用配置方式实现事务管理，配置方式利用代理机制实现，即使有TransactionProxyFactoryBean类来为目标类代理事务管理。

接下来演示一下具体使用吧：

1、重新定义业务类实现，在业务类中无需显示的事务管理代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略import
public class ConfigAddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
    @Override
    public void save(final AddressModel address) {
        addressDao.save(address);
    }
    //countAll方法实现不变
}
```

```
package cn.javass.spring.chapter9.service.impl;
//省略import
public class ConfigUserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Override
    public void save(final UserModel user) {
        userDao.save(user);
        user.getAddress().setUserId(user.getId());
        addressService.save(user.getAddress());
    }
    //countAll方法实现不变
}
```

从以上业务类中可以看出，没有事务管理的代码，即没有侵入式的代码。

2、在**chapter9/service/applicationContext-service.xml**配置文件中添加如下配置：

2.1、首先添加目标类定义：

```
<bean id="targetUserService" class="cn.javass.spring.chapter9.service.impl.ConfigUserServ
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="targetAddressService"/>
</bean>
<bean id="targetAddressService" class="cn.javass.spring.chapter9.service.impl.ConfigAddre
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.2、配置**TransactionProxyFactoryBean**类：

```
<bean id="transactionProxyParent" class="org.springframework.transaction.interceptor.Tran
    <property name="transactionManager" ref="txManager"/>
    <property name="transactionAttributes">
    <props>
            <prop key="save*">
                      PROPAGATION_REQUIRED,
                      ISOLATION_READ_COMMITTED,
                      timeout_10,
                      -Exception,
                      +NoRollBackException
           </prop>
           <prop key="*">
                      PROPAGATION_REQUIRED,
                      ISOLATION_READ_COMMITTED,
                      readOnly
           </prop>
        </props>
</property>
</bean>
```

- **TransactionProxyFactoryBean**：用于为目标业务类创建代理的Bean；
- **abstract="true"**：表示该Bean是抽象的，用于去除重复配置；

- **transactionManager**：事务管理器定义；
- **transactionAttributes**：表示事务属性定义：
- **PROPAGATION_REQUIRED,ISOLATION_READ_COMMITTED,timeout_10,-Exception,+NoRollBackException**：事务属性定义，Required传播行为，提交读隔离级别，事务超时时间为10秒，将对所有Exception异常回滚，而对于抛出NoRollBackException异常将不发生回滚而是提交；
- **PROPAGATION_REQUIRED,ISOLATION_READ_COMMITTED,readOnly**：事务属性定义，Required传播行为，提交读隔离级别，事务是只读的，且只对默认的RuntimeException异常回滚；
- **<prop key="save*">**：表示将代理以save开头的方法，即当执行到该方法时会为该方法根据事务属性配置来开启/关闭事务；
- **<prop key="*">**：表示将代理其他所有方法，但需要注意代理方式，默认是JDK代理，只有public方法能代理；

注：事务属性的传播行为和隔离级别使用TransactionDefinition静态变量名指定；事务超时使用“timeout_超时时间”指定，事务只读使用“readOnly”指定，需要回滚的异常使用“-异常”指定，不需要回滚的异常使用“+异常”指定，默认只对RuntimeException异常回滚。

需要特别注意“-异常”和“+异常”中“异常”只是真实异常的部分名，内部使用如下方式判断：

```
//真实抛出的异常.name.indexOf(配置中指定的需要回滚/不回滚的异常名)
exceptionClass.getName().indexOf(this.exceptionName)
```

因此异常定义时需要特别注意，配置中定义的异常只是真实异常的部分名。

2.3、定义代理**Bean**：

```
<bean id="proxyUserService" parent="transactionProxyParent">
    <property name="target" ref="targetUserService"/>
</bean>
<bean id="proxyAddressService" parent="transactionProxyParent">
    <property name="target" ref="targetAddressService"/>
</bean>
```

代理Bean通过集成抽象Bean“transactionProxyParent”，并通过target属性设置目标Bean，在实际使用中应该使用该代理Bean。

3、修改测试方法并测试该配置方式是否好用：

将TransactionTest 类的testServiceTransaction测试方法拷贝一份命名为testConfigTransaction：

并在testConfigTransaction测试方法内将：

```
IUserService userService =
ctx2.getBean("userService", IUserService.class);
IAddressService addressService =
ctx2.getBean("addressService", IAddressService.class);
```

替换为：

```
IUserService userService =
ctx2.getBean("proxyUserService ", IUserService.class);
IAddressService addressService =
ctx2.getBean("proxyAddressService ", IAddressService.class);
```

4、执行测试，测试正常通过，说明该方式能正常工作，当调用save方法时将匹配到"**<prop key="save*">**"定义，而countAll将匹配到"**<prop key="save*">**"定义，底层代理会应用相应定义中的事务属性来创建或关闭事务。

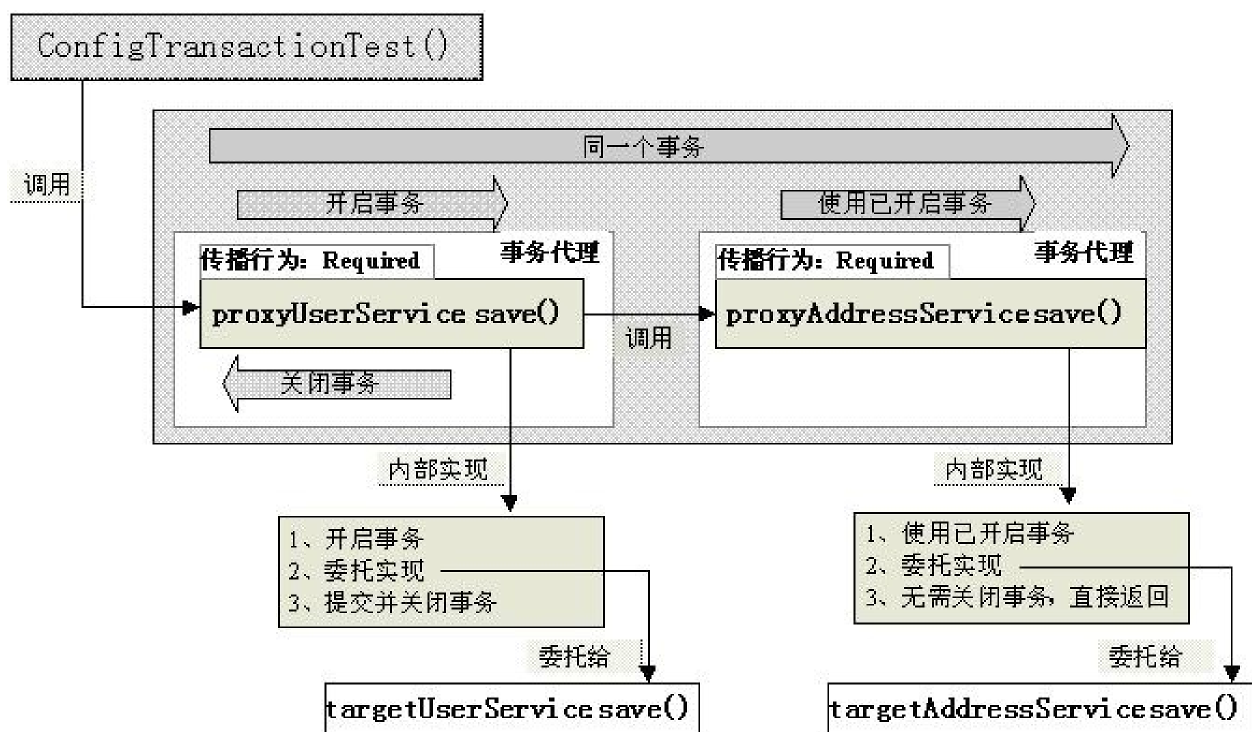


图9-18 代理方式实现事务管理

如图9-18，代理方式实现事务管理只是将硬编码的事务管理代码转移到代理中去由代理实现，在代理中实现事务管理。

注：在代理模式下，默认只有通过代理对象调用的方法才能应用相应的事务属性，而在目标方法内的"自我调用"是不会应用相应的事务属性的，即被调用方法不会应用相应的事务属性，而是使用调用方法的事务属性。

如图9-19所示，在目标对象targetUserService的save方法内调用事务方法"this.otherTransactionMethod()"将不会应用配置的传播行为RequriesNew，开启新事务，而是使用save方法的已开启事务，如果非要这样使用如下方式实现：

1、修改TransactionProxyFactoryBean配置定义，添加exposeProxy属性为true；

2、在业务方法内通过代理对象调用相应的事务方放，如“((IUserService)AopContext.currentProxy()).otherTransactionMethod()”即可应用配置的事务属性。

3、使用这种方式属于侵入式，不推荐使用，除非必要。

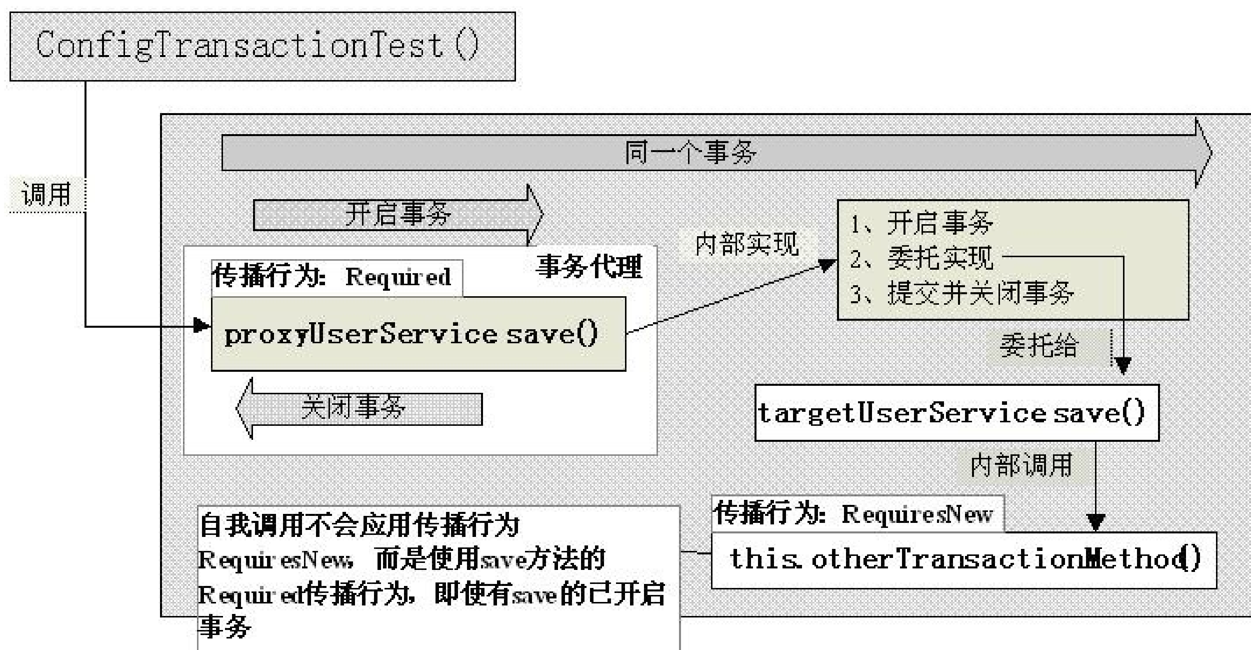


图9-19 代理方式下的自我调用

配置方式也好麻烦啊，每个业务实现都需要配置一个事务代理，发展到这，Spring想出更好的解决方案，Spring2.0及之后版本提出使用新的“<tx:tags/>”方式配置事务，从而无需为每个业务实现配置一个代理。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/2506.html】

【第九章】 Spring的事务 之 9.4 声明式事务 ——跟我学spring3

9.4 声明式事务

9.4.1 声明式事务概述

从上节编程式实现事务管理可以深刻体会到编程式事务的痛苦，即使通过代理配置方式也是不小的工作量。

本节将介绍声明式事务支持，使用该方式后最大的获益是简单，事务管理不再是令人痛苦的，而且此方式属于无侵入式，对业务逻辑实现无影响。

接下来先来看看声明式事务如何实现吧。

9.4.2 声明式实现事务管理

1、定义业务逻辑实现，此处使用**ConfigUserServiceImpl**和**ConfigAddressServiceImpl**：

2、定义配置文件（**chapter9/service/ applicationContext-service-declare.xml**）：

2.1、**XML**命名空间定义，定义用于事务支持的**tx**命名空间和**AOP**支持的**aop**命名空间：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
          http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

2.2、业务实现配置，非常简单，使用以前定义的非侵入式业务实现：

```
<bean id="userService" class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImp
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService" class="cn.javass.spring.chapter9.service.impl.ConfigAddressServ
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.3、事务相关配置：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" isolation="READ_COMMITTED"/>
        <tx:method name="*" propagation="REQUIRED" isolation="READ_COMMITTED" read-only="
    </tx:attributes>
</tx:advice>
```

```
<aop:config>
    <aop:pointcut id="serviceMethod" expression="execution(* cn..chapter9.service..*.*(..
    <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice"/>
</aop:config>
```

- <tx:advice>：事务通知定义，用于指定事务属性，其中“transaction-manager”属性指定事务管理器，并通过< tx:attributes >指定具体需要拦截的方法；
- <tx:method name="save*">：表示将拦截以save开头的方法，被拦截的方法将应用配置的事务属性：propagation="REQUIRED"表示传播行为是Required，isolation="READ_COMMITTED"表示隔离级别是提交读；
- <tx:method name="*">：表示将拦截其他所有方法，被拦截的方法将应用配置的事务属性：propagation="REQUIRED"表示传播行为是Required，isolation="READ_COMMITTED"表示隔离级别是提交读，read-only="true"表示事务只读；

- <aop:config>：AOP相关配置：

- <aop:pointcut/>：切入点定义，定义名为"serviceMethod"的aspectj切入点，切入点表达式为"execution(*cn..chapter9.service...*(..))"表示拦截cn包及子包下的chapter9. service包及子包下的任何类的任何方法；
- <aop:advisor>：Advisor定义，其中切入点为serviceMethod，通知为txAdvice。

从配置中可以看出，将对cn包及子包下的chapter9. service包及子包下的任何类的任何方法应用“txAdvice”通知指定的事务属性。

3、修改测试方法并测试该配置方式是否好用：

将TransactionTest 类的testServiceTransaction测试方法拷贝一份命名为testDeclareTransaction：

并在testDeclareTransaction测试方法内将：

```
classpath:chapter9/service/applicationContext-service.xml"
```

替换为：

```
classpath:chapter9/service/applicationContext-service-declare.xml"
```

4、执行测试，测试正常通过，说明该方式能正常工作，当调用save方法时将匹配到事务通知中定义的“<tx:method name="save">”中指定的事务属性，而调用*countAll*方法时将匹配到事务通知中定义的“*<tx:method name=*"*">”中指定的事务属性。

声明式事务是如何实现事务管理的呢？还记不记得TransactionProxyFactoryBean实现配置式事务管理，配置式事务管理是通过代理方式实现，而声明式事务管理同样是通过AOP代理方式实现。

声明式事务通过AOP代理方式实现事务管理，利用环绕通知TransactionInterceptor实现事务的开启及关闭，而TransactionProxyFactoryBean内部也是通过该环绕通知实现的，因此可以认为是<tx:tags/>帮你定义了TransactionProxyFactoryBean，从而简化事务管理。

了解了实现方式后，接下来详细学习一下配置吧：

9.4.4 <tx:advice/>配置详解

声明式事务管理通过配置<tx:advice/>来定义事务属性，配置方式如下所示：

```
<tx:advice id="……" transaction-manager="……">
<tx:attributes>
        <tx:method name="……"
                           propagation=" REQUIRED"
                           isolation="READ_COMMITTED"
                           timeout="-1"
                           read-only="false"
                           no-rollback-for=""
                           rollback-for=""/>
        ……
    </tx:attributes>
</tx:advice>
```

- **<tx:advice>**：id用于指定此通知的名字， transaction-manager用于指定事务管理器，默认的事务管理器名字为“transactionManager”；
- <tx:method>：用于定义事务属性即相关联的方法名；

name：定义与事务属性相关联的方法名，将对匹配的方法应用定义的事务属性，可以使用“”通配符来匹配一组或所有方法，如“*save*”将匹配以save开头的方法，而“*”将匹配所有方法；

propagation：事务传播行为定义，默认为“REQUIRED”，表示Required，其值可以通过TransactionDefinition的静态传播行为变量的“PROPAGATION_”后边部分指定，如“TransactionDefinition.PROPAGATION_REQUIRED”可以使用“REQUIRED”指定；

isolation：事务隔离级别定义；默认为“DEFAULT”，其值可以通过TransactionDefinition的静态隔离级别变量的“ISOLATION_”后边部分指定，如“TransactionDefinition.ISOLATION_DEFAULT”可以使用“DEFAULT”指定：

timeout：事务超时时间设置，单位为秒，默认-1，表示事务超时将依赖于底层事务系统；

read-only：事务只读设置，默认为false，表示不是只读；

rollback-for：需要触发回滚的异常定义，以“，”分割，默认任何RuntimeException 将导致事务回滚，而任何Checked Exception 将不导致事务回滚；异常名字定义和TransactionProxyFactoryBean中含义一样

no-rollback-for：不被触发进行回滚的 Exception(s)；以“，”分割；异常名字定义和TransactionProxyFactoryBean中含义一样；

记不记得在配置方式中为了解决“自我调用”而导致的不能设置正确的事务属性问题，使用“**((IUserService)AopContext.currentProxy()).otherTransactionMethod()**”方式解决，在声明式事务要得到支持需要使用<aop:config expose-proxy="true">来开启。

9.4.5 多事务语义配置及最佳实践

什么是多事务语义？说白了就是为不同的Bean配置不同的事务属性，因为我们项目中不可能就几个Bean，而可能很多，这可能需要为Bean分组，为不同组的Bean配置不同的事务语义。在Spring中，可以通过配置多切入点和多事务通知并通过不同方式组合使用即可。

1、首先看下声明式事务配置的最佳实践吧：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
<tx:attributes>
           <tx:method name="save*" propagation="REQUIRED" />
           <tx:method name="add*" propagation="REQUIRED" />
           <tx:method name="create*" propagation="REQUIRED" />
           <tx:method name="insert*" propagation="REQUIRED" />
           <tx:method name="update*" propagation="REQUIRED" />
           <tx:method name="merge*" propagation="REQUIRED" />
           <tx:method name="del*" propagation="REQUIRED" />
           <tx:method name="remove*" propagation="REQUIRED" />
           <tx:method name="put*" propagation="REQUIRED" />
           <tx:method name="get*" propagation="SUPPORTS" read-only="true" />
           <tx:method name="count*" propagation="SUPPORTS" read-only="true" />
          <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
          <tx:method name="list*" propagation="SUPPORTS" read-only="true" />
          <tx:method name="*" propagation="SUPPORTS" read-only="true" />
       </tx:attributes>
</tx:advice>
<aop:config>
       <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service.*.*(..))"
       <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
```

该声明式事务配置可以应付常见的CRUD接口定义，并实现事务管理，我们只需修改切入点表达式来拦截我们的业务实现从而对其应用事务属性就可以了，如果还有更复杂的事务属性直接添加即可，即

如果我们有一个batchSaveOrUpdate方法需要“REQUIRES_NEW”事务传播行为，则直接添加如下配置即可：

```
<tx:method name="batchSaveOrUpdate" propagation="REQUIRES_NEW" />
```

2、接下来看一下多事务语义配置吧，声明式事务最佳实践中已经配置了通用事务属性，因此可以针对需要其他事务属性的业务方法进行特例化配置：

```
<tx:advice id="noTxAdvice" transaction-manager="txManager">
    <tx:attributes>
           <tx:method name="*" propagation="NEVER" />
    </tx:attributes>
</tx:advice>
<aop:config>
       <aop:pointcut id="noTxPointcut" expression="execution(* cn.javass..util.*.*())" />
       <aop:advisor advice-ref="noTxPointcut" pointcut-ref="noTxAdvice" />
</aop:config>
```

该声明将对切入点匹配的方法所在事务应用“Never”传播行为。

多事务语义配置时，切入点一定不要叠加，否则将应用两次事务属性，造成不必要的错误及麻烦。

9.4.6 @Transactional实现事务管理

对声明式事务管理，Spring提供基于@Transactional注解方式来实现，但需要Java 5+。

注解方式是最简单的事务配置方式，可以直接在Java源代码中声明事务属性，且对于每一个业务类或方法如果需要事务都必须使用此注解。

接下来学习一下注解事务的使用吧：

1、定义业务逻辑实现：

```
package cn.javass.spring.chapter9.service.impl;
//省略import
public class AnnotationUserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Transactional(propagation=Propagation.REQUIRED, isolation=Isolation.READ_COMMITTED)
    @Override
    public void save(final UserModel user) {
        userDao.save(user);
        user.getAddress().setUserId(user.getId());
        addressService.save(user.getAddress());
    }
    @Transactional(propagation=Propagation.REQUIRED, isolation=Isolation.READ_COMMITTED,
    @Override
    public int countAll() {
        return userDao.countAll();
    }
}
```

2、定义配置文件（**chapter9/service/ applicationContext-service-annotation.xml**）：

2.1、**XML**命名空间定义，定义用于事务支持的**tx**命名空间和**AOP**支持的**aop**命名空间：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
          http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

2.2、业务实现配置，非常简单，使用以前定义的非侵入式业务实现：

```
<bean id="userService" class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImp
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService" class="cn.javass.spring.chapter9.service.impl.ConfigAddressServ
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.3、事务相关配置：

```
<tx:annotation-driven transaction-manager="txManager"/>
```

使用如上配置已支持声明式事务。

3、修改测试方法并测试该配置方式是否好用：

将TransactionTest 类的testServiceTransaction测试方法拷贝一份命名为testAnntationTransactionTest：

将测试代码片段：

```
classpath:chapter9/service/applicationContext-service.xml"
```

替换为：

```
classpath:chapter9/service/applicationContext-service-annotation.xml"
```

将测试代码段

```
userService.save(user);
```

替换为：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(0, addressService.countAll());
```

4、执行测试，测试正常通过，说明该方式能正常工作，因为在AnnotationAddressServiceImpl类的save方法中抛出异常，因此事务需要回滚，所以两个countAll操作都返回0。

9.4.7 @Transactional配置详解

Spring提供的<tx:annotation-driven/>用于开启对注解事务管理的支持，从而能识别Bean类上的@Transactional注解元数据，其具有以下属性：

- transaction-manager：指定事务管理器名字，默认为transactionManager，当使用其他名字时需要明确指定；
- proxy-target-class：表示将使用的代码机制，默认false表示使用JDK代理，如果为true将使用CGLIB代理
- order：定义事务通知顺序，默认Ordered.LOWEST_PRECEDENCE，表示将顺序决定权交给AOP来处理。

Spring使用@Transaction来指定事务属性，可以在接口、类或方法上指定，如果类和方法上都指定了@Transaction，则方法上的事务属性被优先使用，具体属性如下：

- value：指定事务管理器名字，默认使用<tx:annotation-driven/>指定的事务管理器，用于支持多事务管理器环境；
- **propagation**：指定事务传播行为，默认为Required，使用Propagation.REQUIRED指定；
- **isolation**：指定事务隔离级别，默认为"DEFAULT"，使用Isolation.DEFAULT指定；
- **readOnly**：指定事务是否只读，默认false表示事务非只读；
- **timeout**：指定事务超时时间，以秒为单位，默认-1表示事务超时将依赖于底层事务系统；
- **rollbackFor**：指定一组异常类，遇到该类异常将回滚事务；
- **rollbackForClassname**：指定一组异常类名字，其含义与<tx:method>中的rollback-for属性语义完全一样；
- **noRollbackFor**：指定一组异常类，即使遇到该类异常也将提交事务，即不回滚事务；
- **noRollbackForClassname**：指定一组异常类名字，其含义与<tx:method>中的no-rollback-for属性语义完全一样；

Spring提供的@Transaction注解事务管理内部同样利用环绕通知TransactionInterceptor实现事务的开启及关闭。

使用@Transactional注解事务管理需要特别注意以下几点：

- 如果在接口、实现类或方法上都指定了@Transactional 注解，则优先级顺序为方法>实现类>接口；
- 建议只在实现类或实现类的方法上使用@Transactional，而不要在接口上使用，这是因为如果使用JDK代理机制是没问题，因为其使用基于接口的代理；而使用使用CGLIB代理机制时就会遇到问题，因为其使用基于类的代理而不是接口，这是因为接口上的**@Transactional**注解是"不能继承的"；

具体请参考基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。

- 在Spring代理机制下(不管是JDK动态代理还是CGLIB代理)，"自我调用"同样不会应用相应的事务属性，其语义和<tx:tags>中一样；
- 默认只对RuntimeException异常回滚；
- 在使用Spring代理时，默认只有在public可见度的方法的@Transactional 注解才是有效的，其它可见度（protected、private、包可见）的方法上即使有@Transactional 注解也不会应用这些事务属性的，Spring也不会报错，如果你非要使用非公共方法注解事务管理的话，可考虑使用AspectJ。

9.4.9 与其他AOP通知协作

Spring声明式事务实现其实就是Spring AOP+线程绑定实现，利用AOP实现开启和关闭事务，利用线程绑定（ThreadLocal）实现跨越多个方法实现事务传播。

由于我们不可能只使用一个事务通知，可能还有其他类型事务通知，而且如果这些通知中需要事务支持怎么办？这就牵扯到通知执行顺序的问题上了，因此如果可能与其他AOP通知协作的话，而且这些通知中需要使用声明式事务管理支持，事务通知应该具有最高优先级。

9.4.10 声明式or编程式

编程式事务时不推荐的，即使有很少事务操作，Spring发展到现在，没有理由使用编程式事务，只有在为了深入理解Spring事务管理才需要学习编程式事务使用。

推荐使用声明式事务，而且强烈推荐使用<tx:tags>方式的声明式事务，因为其是无侵入代码的，可以配置模板化的事务属性并运用到多个项目中。

而@Transaction注解事务，可以使用，不过作者更倾向于使用<tx:tags>声明式事务。

能保证项目正常工作的事务配置就是最好的。

9.4.11 混合事务管理

所谓混合事务管理就是混合多种数据访问技术使用，如混合使用Spring JDBC + Hibernate，接下来让我们学习一下常见混合事务管理：

1、 Hibernate + Spring JDBC/iBATIS：使用HibernateTransactionManager即可支持；

2、 JPA + Spring JDBC/iBATIS：使用JpaTransactionManager即可支持；

3、 JDO + Spring JDBC/iBATIS：使用JtaTransactionManager即可支持；

混合事务管理最大问题在于如果我们使用第三方ORM框架，如Hibernate，会遇到一级及二级缓存问题，尤其是二级缓存可能造成如使用Spring JDBC和Hibernate查询出来的数据不一致等。

因此不建议使用这种混合使用和混合事务管理。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2508.html】

【第十章】集成其它Web框架 之 10.1 概述 ——跟我学spring3

10.1 概述

10.1.1 Spring和Web框架

Spring框架不仅提供了一套自己的Web框架实现，还支持集成第三方Web框架（如Struts1x、Struts2x）。

Spring实现的SpringMVC Web框架将在第十八章详细介绍。

由于现在有很大部分公司在使用第三方Web框架，对于并不熟悉SpringMVC Web框架的公司，为了充分利用开发人员已掌握的技术并相使用Spring的功能，想集成所使用的Web框架；由于Spring框架的高度可配置和可选择性，因此集成这些第三方Web框架是非常简单的。

之所以想把这些第三方Web框架集成到Spring中，最核心的价值是享受Spring的某些强大功能，如一致的数据访问，事务管理，IOC，AOP等等。

Spring为所有Web框架提供一致的通用配置，从而不管使用什么Web框架都使用该通用配置。

10.1.2 通用配置

Spring对所有Web框架抽象出通用配置，以减少重复配置，其中主要有以下配置：

1、Web环境准备：

1.1、在spring项目下创建如图10-1目录结构：

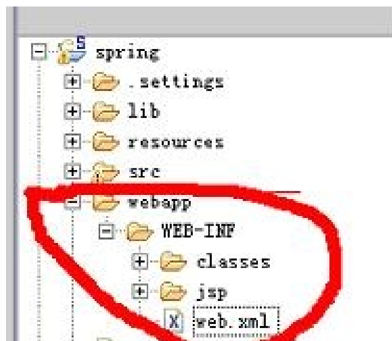


图10-1 web目录结构

1.2、右击**spring**项目选择【**Propeties**】，然后选择【**Java Build Path**】中的【**Source**】选项卡，将类输出路径修改为**“spring/webapp/WEB-INF/classes”**，如图**10-2**所示：

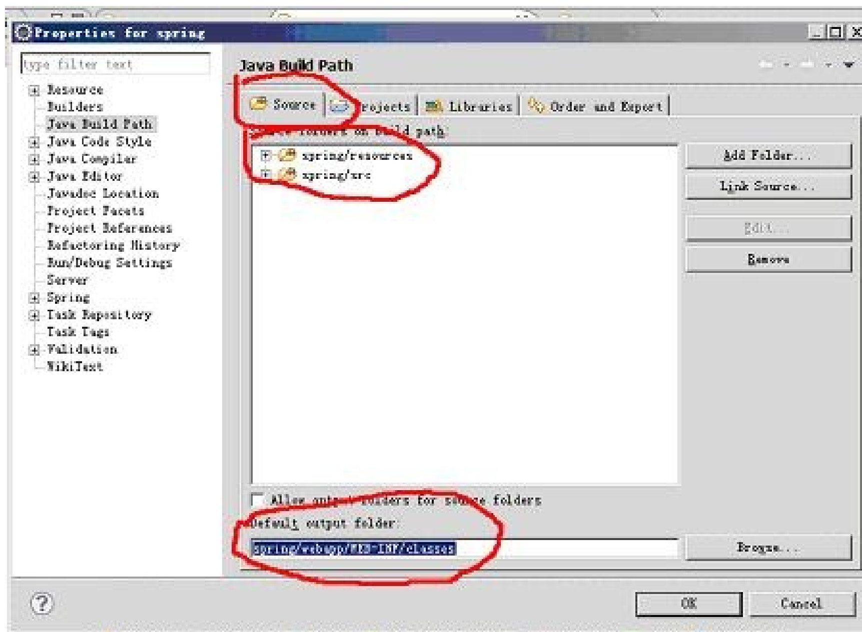


图10-2 修改类输出路径

1.3、**web.xml**初始内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
       http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
</web-app>
```

<web-app version="2.4">表示采用Servlet 2.4规范的Web程序部署描述格式

2、 指定**Web**应用上下文实现：在Web环境中，Spring提供WebApplicationContext（继承ApplicationContext）接口用于配置Web应用，该接口应该被实现为在Web应用程序运行时只读，即在初始化完毕后不能修改Spring Web容器（WebApplicationContext），但可能支持重载。

Spring提供XmlWebApplicationContext实现，并在Web应用程序中默认使用该实现，可以通过在web.xml配置文件中使用如下方式指定：

```
<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.XmlWebApplicationContext
    </param-value>
</context-param>
```

如上指定是可选的，只有当使用其他实现时才需要显示指定。

3、指定加载文件位置：

前边已经指定了Spring Web容器实现，那从什么地方加载配置文件呢？

默认情况下将加载/WEB-INF/applicationContext.xml配置文件，当然也可以使用如下形式在web.xml中定义要加载自定义的配置文件，多个配置文件用“，”分割：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:chapter10/applicationContext-message.xml
    </param-value>
</context-param>
```

通用Spring配置文件（resources/chapter10/applicationContext-message.xml）内容如下所示：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

4、加载和关闭**Spring Web**容器：

我们已经指定了Spring Web容器实现和配置文件，那如何才能让Spring使用相应的Spring Web容器实现加载配置文件呢？

Spring使用ContextLoaderListener监听器来加载和关闭Spring Web容器，即使用如下方式在web.xml中指定：

```
<listener>
    <listener-class>
       org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener监听器将在Web应用启动时使用指定的配置文件初始化Spring Web容器，在Web应用关闭时销毁Spring Web容器。

注：监听器是从Servlet 2.3才开始支持的，因此如果Web应用所运行的环境是Servlet 2.2版本则可以使用ContextLoaderServlet来完成，但从Spring3.x版本之后ContextLoaderServlet被移除了。

5、在**Web**环境中获取**Spring Web**容器**:**

既然已经定义了Spring Web容器，那如何在Web中访问呢？Spring提供如下方式来支持获取Spring Web容器（WebApplicationContext）：

```
WebApplicationContextUtils.getWebApplicationContext(servletContext);
或
WebApplicationContextUtils.getRequiredWebApplicationContext(servletContext);
```

如果当前Web应用中的ServletContext 中没有相应的Spring Web容器，对于getWebApplicationContext()方法将返回null,而getRequiredWebApplicationContext()方法将抛出异常，建议使用第二种方式，因为缺失Spring Web容器而又想获取它，很明显是错误的，应该抛出异常。

6、 通用**jar**包，从下载的**spring-framework-3.0.5.RELEASE-with-docs.zip**中**dist**目录查找如下**jar**包：

- org.springframework.web-3.0.5.RELEASE.jar

此jar包为所有Web框架所共有，提供WebApplicationContext及实现等。

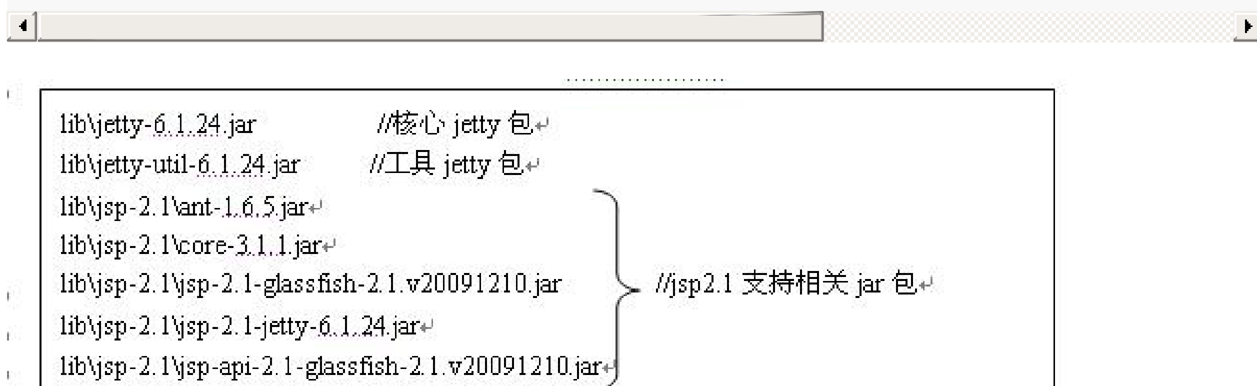
7、**Web**服务器选择及测试：

目前比较流行的支持Servlet规范的开源Web服务器包括Tomcat、Resin、Jetty等，Web服务器有独立运行和嵌入式运行之分，嵌入式Web服务器可以在测试用例中运行不依赖于外部环境，因此我们使用嵌入式Web服务器。

Jetty是一个非常轻量级的Web服务器，并且提供嵌入式运行支持，在此我们选用Jetty作为测试使用的Web服务器。

7.1、准备**Jetty**嵌入式**Web**服务器运行需要的**jar**包：

```
到http://dist.codehaus.org/jetty/网站下载jetty-6.1.24，在下载的jetty-6.1.24.zip包中拷贝如下jar
```



```
lib\jetty-6.1.24.jar              //核心 jetty 包
lib\jetty-util-6.1.24.jar         //工具 jetty 包
lib\jsp-2.1\ant-1.6.5.jar
lib\jsp-2.1\core-3.1.1.jar
lib\jsp-2.1\jsp-2.1-glassfish-2.1.v20091210.jar      //jsp2.1 支持相关 jar 包
lib\jsp-2.1\jsp-2.1-jetty-6.1.24.jar
lib\jsp-2.1\jsp-api-2.1-glassfish-2.1.v20091210.jar
```

7.2、在单元测试中启动**Web**服务器：

```
package cn.javass.spring.chapter10;
import org.junit.Test;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.webapp.WebAppContext;
public class WebFrameWorkIntegrateTest {
    @Test
    public void testWebFrameWork() throws Exception {
        Server server = new Server(8080);
        WebAppContext webapp = new WebAppContext();
        webapp.setResourceBase("webapp");
        //webapp.setDescriptor("webapp/WEB-INF/web.xml");
        webapp.setContextPath("/");
        webapp.setClassLoader(Thread.currentThread().getContextClassLoader());
        server.setHandler(webapp);
        server.start();
        server.join();
       //server.stop();
    }
}
```

- 创建内嵌式**Web**服务器：使用new Server(8080)新建一个Jetty服务器，监听端口为8080；
- 创建一个**Web**应用：使用new WebAppContext()新建一个Web应用对象，一个Web应用可以认为就是一个WebAppContext对象；
- 指定**Web**应用的目录：使用webapp.setResourceBase("webapp")指定Web应用位于项目根目录下的“webapp”目录下；
- 指定部署描述符：使用webapp.setDescriptor("webapp/WEB-INF/web.xml")；此处指定部署描述符为项目根目录下的“webapp/WEB-INF/web.xml”，该步骤是可选的，如果web.xml位于Web应用的WEB-INF下。
- 指定**Web**应用请求上下文：使用webapp.setContextPath("/")指定请求上下文为“/”，从而访问该Web应用可以使用如“http://localhost:8080/hello.do”形式访问；
- 指定类装载器：因为Jetty自带的ClassLoader在内嵌环境中对中文路径处理有问题，因此我们使用Eclispe的ClassLoader，即通过“webapp.setClassLoader(Thread.currentThread().getContextClassLoader()) ”指定；
- 启动**Web**服务器：使用“server.start()”启动并使用“server.join()”保证Web服务器一直运行；
- 关闭**Web**服务器：可以通过某种方式执行“server.stop()”来关闭Web服务器；另一种方式是通过【Console】控制台面板的【Terminate】终止按钮关闭，如图10-3所示：

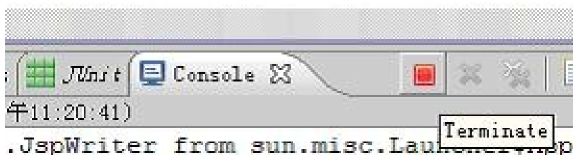


图10-3 点击红色按钮关闭Web服务器

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2510.html】

【第十章】集成其它Web框架 之 10.2 集成Struts1.x ——跟我学spring3

先进行通用配置，【第十章】集成其它Web框架 之 10.1 概述

10.2 集成Struts1.x

10.2.1 概述

Struts1.x是最早实现MVC（模型-视图-控制器）模式的Web框架之一，其使用非常广泛，虽然目前已经有Struts2.x等其他Web框架，但仍有很多公司使用Struts1.x框架。

集成Struts1.x也非常简单，除了通用配置外，有两种方式可以将Struts1.x集成到Spring中：

- 最简单集成：使用Spring提供的WebApplicationContextUtils工具类中的获取Spring Web容器，然后通过Spring Web容器获取Spring管理的Bean；
- Struts1.x插件集成：利用Struts1.x中的插件ContextLoaderPlugin来将Struts1.x集成到Spring中。

接下来让我们首先让我们准备**Struts1x**所需要的**jar**包：

1.1、从下载的**spring-framework-3.0.5.RELEASE-with-docs.zip**中**dist**目录查找如下**jar**包，该**jar**包用于提供集成**struts1.x**所需要的插件实现等：

- org.springframework.web.struts-3.0.5.RELEASE.jar

1.2、从下载的**spring-framework-3.0.5.RELEASE-dependencies.zip**中查找如下依赖**jar**包，该组**jar**是**struts1.x**需要的**jar**包：

- com.springsource.org.apache.struts-1.2.9.jar //struts1.2.9实现包
- com.springsource.org.apache.commons.digester-1.8.1.jar //用于解析struts配置文件
- com.springsource.org.apache.commons.beanutils-1.8.0.jar //用于请求参数绑定
- com.springsource.javax.servlet-2.5.0.jar //Servlet 2.5 API
- antlr.jar //语法分析包（已有）
- commons-logging.jar //日志记录组件包（已有）
- servlet-api.jar //Servlet API包（已有）
- jsp-api.jar //JSP API包（已有，可选）
- **commons-validator.jar //**验证包（可选）
- **commons-fileupload.jar //**文件上传包（可选）

10.2.2 最简单集成

只使用通用配置，利用WebApplicationContextUtils提供的获取Spring Web容器方法获取Spring Web容器，然后从Spring Web容器获取Spring管理的Bean。

1、 第一个Action实现：

```
package cn.javass.spring.chapter10.struts1x.action;
import org.apache.struts.action.Action;
//省略部分import
public class HelloWorldAction1 extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletReque

        WebApplicationContext ctx = WebApplicationContextUtils.
            getRequiredWebApplicationContext(getServlet().getServletContext());
        String message = ctx.getBean("message", String.class);
        request.setAttribute("message",  message);
        return mapping.findForward("hello");
    }
}
```

此Action实现非常简单，首先通过WebApplicationContextUtils获取Spring Web容器，然后从Spring Web容器中获取"message"Bean并将其放到request里，最后转到"hello"所代表的jsp页面。

2、JSP页面定义（webapp/WEB-INF/jsp/hello.jsp）：

```
<%@ page language="java" pageEncoding="UTF-8"
contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    ${message}
</body>
</html>
```

3、配置文件定义：

3.1、Spring配置文件定义（resources/chapter10/applicationContext-message.xml）：

在此配置文件中定义我们使用的"message"Bean；

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

3.2、struts配置文件定义（resources/chapter10/struts1x/struts-config.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <action-mappings>
    <action path="/hello" type="cn.javass.spring.chapter10.struts1x.action.HelloWorldActi
       <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

3.3、**web.xml**部署描述符文件定义（**webapp/WEB-INF/web.xml**）添加如下内容：

```
<!-- Struts1.x前端控制器配置开始   -->
       <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

        <init-param>
            <param-name>config</param-name>
            <param-value>
                      /WEB-INF/classes/chapter10/struts1x/struts-config.xml
             </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
       </servlet>
       <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>*.do</url-pattern>
      </servlet-mapping>
<!-- Struts1.x前端控制器配置结束   -->
```

Struts1.x前端控制器配置了ActionServlet前端控制器，其拦截以.do开头的请求，Strut配置文件通过初始化参数“config”来指定，如果不知道“config”参数则默认加载的配置文件为“/WEB-INF/ struts-config.xml”。

4、执行测试：在Web浏览器中输入http://localhost:8080/hello.do可以看到“Hello Spring”信息说明测试正常。

有朋友想问，我不想使用这种方式，我想在独立环境内测试，没关系，您只需将spring/lib目录拷贝到spring/webapp/WEB-INF/下，然后将webapp拷贝到如tomcat中即可运行，尝试一下吧。

Spring还提供ActionSupport类来简化获取WebApplicationContext，Spring为所有标准Action类及子类提供如下支持类，即在相应Action类后边加上Support后缀：

- ActionSupport
- DispatchActionSupport
- LookupDispatchActionSupport
- MappingDispatchActionSupport

具体使用方式如下：

1、Action定义

```
package cn.javass.spring.chapter10.struts1x.action;
//省略import
public class HelloWorldAction2 extends ActionSupport {
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest r
        WebApplicationContext ctx = getWebApplicationContext();
        String message = ctx.getBean("message", String.class);
        request.setAttribute("message", message);
        return mapping.findForward("hello");
    }
}
```

和第一个示例唯一不同的是直接调用**getWebApplicationContext()**即可获得Spring Web容器。

2、修改**Struts**配置文件（**resources/chapter10/struts1x/struts-config.xml**）添加如下**Action**定义：

```
<action path="/hello2" type="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction2
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

3、启动嵌入式**Web**服务器并在Web浏览器中输入http://localhost:8080/hello2.do可以看到“Hello Spring”信息说明Struts1集成成功。

这种集成方式好吗？而且这种方式算是集成吗？直接获取Spring Web容器然后从该Spring Web容器中获取Bean，暂且看作是集成吧，这种集成对于简单操作可以接受，但更复杂的注入呢？接下来让我们学习使用Struts插件进行集成。

10.2.2 Struts1.x插件集成

Struts插件集成使用ContextLoaderPlugin类，该类用于为ActionServlet加载Spring配置文件。

1、在**Struts**配置文件（**resources/chapter10/struts1x/struts-config.xml**）中配置插件：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextClass" value="org.springframework.web.context.support.
    <set-property property="contextConfigLocation" value="/WEB-INF/hello-servlet.xml"/>
    <set-property property="namespace" value="hello"/>
</plug-in>
```

- **contextClass**：可选，用于指定WebApplicationContext实现类，默认是XmlWebApplicationContext；
- **contextConfigLocation**：指定Spring配置文件位置，如果我们的ActionServlet 在web.xml 里面通过 <servlet-name>hello</servlet-name>指定名字为“hello”，且没有指定contextConfigLocation，则默认Spring配置文件是/WEB-INF/hello-servlet.xml；

- **namespace**：因为默认使用ActionServlet在web.xml定义中的Servlet的名字，因此如果想要使用其他名字可以使用该变量指定，如指定“hello”，将加载的Spring配置文件为/WEB-INF/hello-servlet.xml；

由于我们的ActionServlet在web.xml中的名字为hello，而我们的配置文件在/WEB-INF/hello-servlet.xml，因此contextConfigLocation和namespace可以不指定，因此最简单配置如下：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

通用配置的Spring Web容器将作为ContextLoaderPlugin中创建的Spring Web容器的父容器存在，然而可以省略通用配置而直接在struts配置文件中通过ContextLoaderPlugin插件指定所有配置文件。

插件已经配置了，那如何定义Action、配置Action、配置Spring管理Bean呢，即如何真正集成Spring+Struts1x呢？使用插件方式时Action将在Spring中配置而不是在Struts中配置了，Spring目前提供以下两种方式：

- 将Struts配置文件中的<action>的type属性指定为DelegatingActionProxy，然后在Spring中配置同名的Spring管理的Action Bean；
- 使用Spring提供的DelegatingRequestProcessor重载 Struts 默认的 RequestProcessor来从Spring容器中查找同名的Spring管理的Action Bean。

看懂了吗？好像没怎么看懂，那就直接上代码，有代码有真相。

2、定义**Action**实现，由于**Action**将在**Spring**中配置，因此**message**可以使用依赖注入方式了：

```
package cn.javass.spring.chapter10.struts1x.action;
//省略
public class HelloWorldAction3 extends Action {
    private String message;
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletReque

        request.setAttribute("message", message);
        return mapping.findForward("hello");
    }
    public void setMessage(String message) {//有setter方法，大家是否想到setter注入
        this.message = message;
    }
}
```

3、**DelegatingActionProxy**方式与**Spring**集成配置：

3·**1**、在**Struts**配置文件（**resources/chapter10/struts1x/struts-config.xml**）中进行**Action**定义：

```
<action path="/hello3" type="org.springframework.web.struts.DelegatingActionProxy">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

3.2、在**Spring**配置文件（**webapp/WEB-INF/hello-servlet.xml**）中定义**Action**对应的**Bean**：

```
<bean name="/hello3" class="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3"
    <property name="message" ref="message"/>
</bean>
```

3.3、启动嵌入式**Web**服务器并在Web浏览器中输入http://localhost:8080/hello3.do可以看到“Hello Spring”信息说明测试正常。

从以上配置中可以看出：

- Struts配置文件中<action>标签的path属性和Spring配置文件的name属性应该完全一样，否则错误；
- Struts通过**DelegatingActionProxy**去到Spring Web容器中查找同名的Action Bean；

很简单吧，DelegatingActionProxy是个代理Action，其实现了Action类，其内部帮我们查找相应的Spring管理Action Bean并把请求转发给这个真实的Action。

4、**DelegatingRequestProcessor**方式与**Spring**集成：

4.1、首先要替换掉**Struts**默认的**RequestProcessor**，在**Struts**配置文件（**resources/chapter10/struts1x/struts-config.xml**）中添加如下配置：

```
<controller>
    <set-property property="processorClass"
         value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

4·**2**、在**Struts**配置文件（**resources/chapter10/struts1x/struts-config.xml**）中进行**Action**定义：

```
<action path="/hello4" type=" cn.javass.spring.chapter10.struts1x.action.HelloWorldAction
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

或更简单形式：

```
<action path="/hello4">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

4.3、在**Spring**配置文件（**webapp/WEB-INF/hello-servlet.xml**）中定义**Action**对应的**Bean**：

```
<bean name="/hello4" class="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3"
    <property name="message" ref="message"/>
</bean>
```

4.4、启动嵌入式**Web**服务器并在Web浏览器中输入http://localhost:8080/hello4.do可以看到“Hello Spring”信息说明Struts1集成成功。

从以上配置中可以看出：

- Struts配置文件中<action>标签的path属性和Spring配置文件的name属性应该完全一样，否则错误；
- Struts通过**DelegatingRequestProcessor**去到Spring Web容器中查找同名的Action Bean；

很简单吧，只是由**DelegatingRequestProcessor**去帮我们查找相应的Action Bean，但没有代理Action了，所以推荐使用该方式。

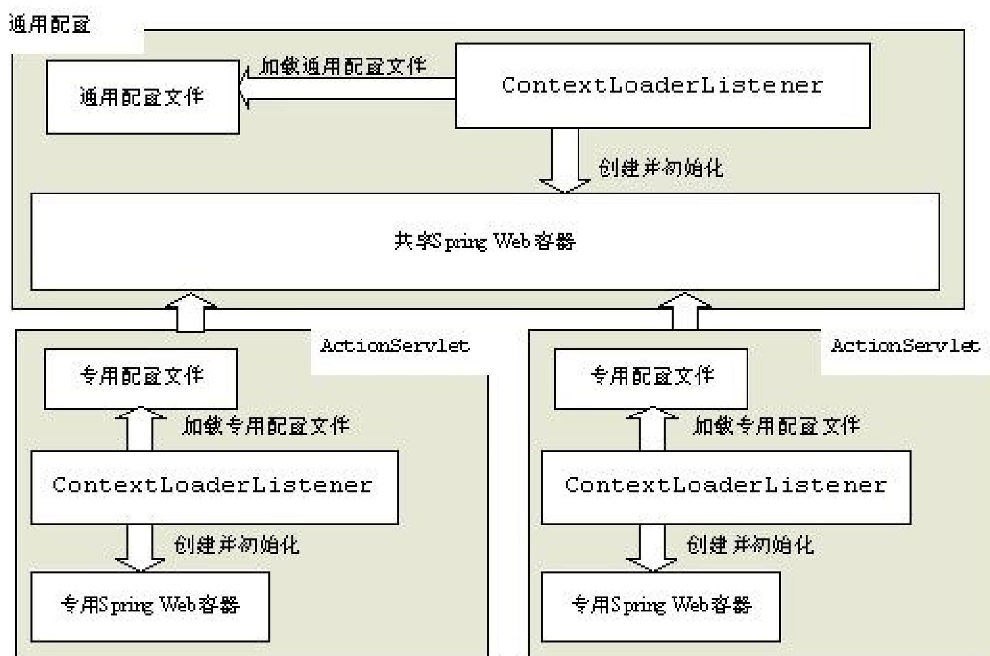


图10-4 共享及专用Spring Web容器

Struts1x与Spring集成到此就完成了，在集成时需要注意一下几点：

- 推荐使用ContextLoaderPlugin+DelegatingRequestProcessor方式集成；
- 当有多个Struts模块时建议在通用配置部分配置通用部分，因为通用配置在正在Web容器中是可共享的，而在各个Struts模块配置文件中配置是不可共享的，因此不推荐直接使用ContextLoaderPlugin中为每个模块都指定所有配置，因为**ContextLoaderPlugin**加载的**Spring**容器只对当前的**ActionServlet**有效对其他**ActionServlet**无效，如图10-4所示。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/2511.html】

【第十章】集成其它Web框架 之 10.3 集成Struts2.x ——跟我学spring3

先进行通用配置，【第十章】集成其它Web框架 之 10.1 概述

10.3 集成Struts2.x

10.3.1 概述

Struts2前身是WebWork，核心并没有改变，其实就是把WebWork改名为struts2，与Struts1一点关系没有。

Struts2中通过ObjectFactory接口实现创建及获取Action实例，类似于Spring的IoC容器，所以Action实例可以由ObjectFactory实现来管理，因此集成Spring的关键点就是如何创建ObjectFactory实现来从Spring容器中获取相应的Action Bean。

Struts2提供一个默认的ObjectFactory接口实现StrutsSpringObjectFactory，该类用于根据Struts2配置文件中相应Bean信息从Spring 容器中获取相应的Action。

因此Struts2.x与Spring集成需要使用StrutsSpringObjectFactory类作为中介者。

接下来让我们首先让我们准备**Struts2x**所需要的**jar**包

准备**Struts2.x**需要的**jar**包，到**Struts**官网**http://struts.apache.org/**下载**struts-2.2.1.1**版本，拷贝如下**jar**包到项目的**lib**目录下并添加到类路径：

- lib\struts2-core-2.2.1.1.jar //核心struts2包
- lib\xwork-core-2.2.1.1.jar //命令框架包，独立于Web环境，为Struts2
- //提供核心功能的支持包
- lib\freemarker-2.3.16.jar //提供模板化UI标签及视图技术支持
- lib\ognl-3.0.jar //对象图导航工具包，类似于SpEL
- lib\ struts2-spring-plugin-2.2.1.1.jar //集成Spring的插件包
- lib\commons-logging-1.0.4.jar //日志记录组件包（已有）
- lib\commons-fileupload-1.2.1.jar //用于支持文件上传的包

10.3.2 使用ObjectFactory集成

1、Struts2.x的**Action**实现：

```
package cn.javass.spring.chapter10.struts2x.action;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class HelloWorldAction extends ActionSupport {
    private String message;
    @Override
    public String execute() throws Exception {
        ServletActionContext.getRequest().setAttribute("message", message);
        return "hello";
    }
    public void setMessage(String message) {//setter注入
        this.message = message;
    }
}
```

2、**JSP**页面定义，使用**Struts1x**中定义的**JSP**页面**“webapp/WEB-INF/jsp/hello.jsp”**；

3、**Spring**一般配置文件定义（**resources/chapter10/applicationContext-message.xml**）：

在此配置文件中定义我们使用的“message”Bean；

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

4、**Spring Action** 配置文件定义（**resources/chapter10/hello-servlet.xml**）：

```
<bean name="helloAction" class="cn.javass.spring.chapter10.struts2x.action.HelloWorldActi
    <property name="message" ref="message"/>
</bean>
```

Struts2的Action在Spring中配置，而且应该是prototype，因为Struts2的Action是有状态的，定义在Spring中，那Struts如何找到该Action呢？

5、**struts2**配置文件定义（**resources/chapter10/struts2x/struts.xml**）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.objectFactory" value="org.apache.struts2.spring.StrutsSpringOb
    <constant name="struts.devMode" value="true"/>
    <package name="default" extends="struts-default">
        <action name="hello" class="helloAction">
            <result name="hello" >/WEB-INF/jsp/hello.jsp</result>
        </action>
    </package>
</struts>
```

- **struts.objectFactory**：通过在Struts配置文件中使用常量属性**struts.objectFactory**来定

义Struts将要使用的ObjectFactory实现，此处因为需要从Spring容器中获取Action对象，因此需要使用StrutsSpringObjectFactory来集成Spring；

- <action name="hello" **class="helloAction"**>：StrutsSpringObjectFactory对象工厂将根据<action>标签的class属性去Spring容器中查找同名的Action Bean；即本例中将到Spring容器中查找名为helloAction的Bean。

6、**web.xml**部署描述符文件定义（**webapp/WEB-INF/web.xml**）：

6.1、由于**Struts2**只能使用通用配置，因此需要在通用配置中加入**Spring Action**配置文件（**chapter10/struts2x/struts2x-servlet.xml**）：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:chapter10/applicationContext-message.xml,
        classpath:chapter10/struts2x/struts2x-servlet.xml
    </param-value>
</context-param>
```

Struts2只能在通用配置中指定所有Spring配置文件，并没有如Struts1自己指定Spring配置文件的实现。

6.2、**Strut2**前端控制器定义，在**web.xml**中添加如下配置：

```
<!-- Struts2.x前端控制器配置开始   -->
<filter>
    <filter-name>struts2x</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    <init-param>
              <param-name>config</param-name>
              <param-value>
                     struts-default.xml,struts-plugin.xml,chapter10/struts2x/struts.xml
               </param-value>
       </init-param>
</filter>
<filter-mapping>
    <filter-name>struts2x</filter-name>
    <url-pattern>*.action</url-pattern>
</filter-mapping>
<!-- Struts2.x前端控制器配置结束   -->
```

- **FilterDispatcher**：Struts2前端控制器为FilterDispatcher，是Filter实现，不是Servlet；
- **config**：通过初始化参数config指定配置文件为struts-default.xml，struts-plugin.xml，chapter10/struts2x/struts.xml；如果不知道该参数则默认加载struts-default.xml，struts-plugin.xml，struts.xml(位于webapp/WEB-INF/classes下)；显示指定时需要将struts-default.xml，struts-plugin.xml也添加上。
- ***.action**：将拦截以“.action”结尾的HTTP请求；
- **struts2x**：FilterDispatcher前端控制器的名字为struts2x，因此相应的Spring配置文件名为struts2x-servlet.xml。

7、执行测试，在**Web**浏览器中输入**http://localhost:8080/hello.action**可以看到**"Hello Spring"**信息说明**Struts2**集成成功。

集成**Strut2**也是非常简单，在此我们总结一下吧：

- 配置文件位置：

Struts配置文件默认加载"struts-default.xml,struts-plugin.xml, struts.xml"，其中struts-default.xml和struts-plugin.xml是Struts自带的，而struts.xml是我们指定的，默认位于webapp/WEB-INF/classes下；

如果需要将配置文件放到其他位置，需要在web.xml的<filter>标签下，使用初始化参数config指定，如"struts-default.xml，struts-plugin.xml，chapter10/struts2x/struts.xml"，其中"struts-default.xml和struts-plugin.xml"是不可省略的，默认相对路径是类路径。

- 集成关键**ObjectFactory**：在Struts配置文件或属性文件中使用如下配置知道使用StrutsSpringObjectFactory来获取Action实例：

在struts.xml中指定：

```
<constant name="struts.objectFactory" value="org.apache.struts2.spring.StrutsSpringObject
```

或在**struts.properties**文件（**webapp/WEB-INF/classes/**）中：

```
struts.objectFactory=org.apache.struts2.spring.StrutsSpringObjectFactory
```

- 集成关键**Action**定义：

StrutsSpringObjectFactory将根据Struts2配置文件中的<action class="">标签的classes属性名字去到Spring配置文件中查找同名的Bean定义，这也是集成的关键。

```
<action name="hello" class="helloAction">
      ......
</action>
```

通过同名来集成

```
<bean name="helloAction" class="Action类全限定名" scope="prototype">
      ......
</bean>
```

- **Spring**配置文件中**Action**定义：由于Struts2的Action是有状态的，因此应该将Bean定义为prototype。

如图10-5，Sturt2与Spring集成的关键就是StrutsSpringObjectFactory，注意图只是说明Struts与Spring如何通过中介者StrutsSpringObjectFactory来实现集成，不能代表实际的类交互。

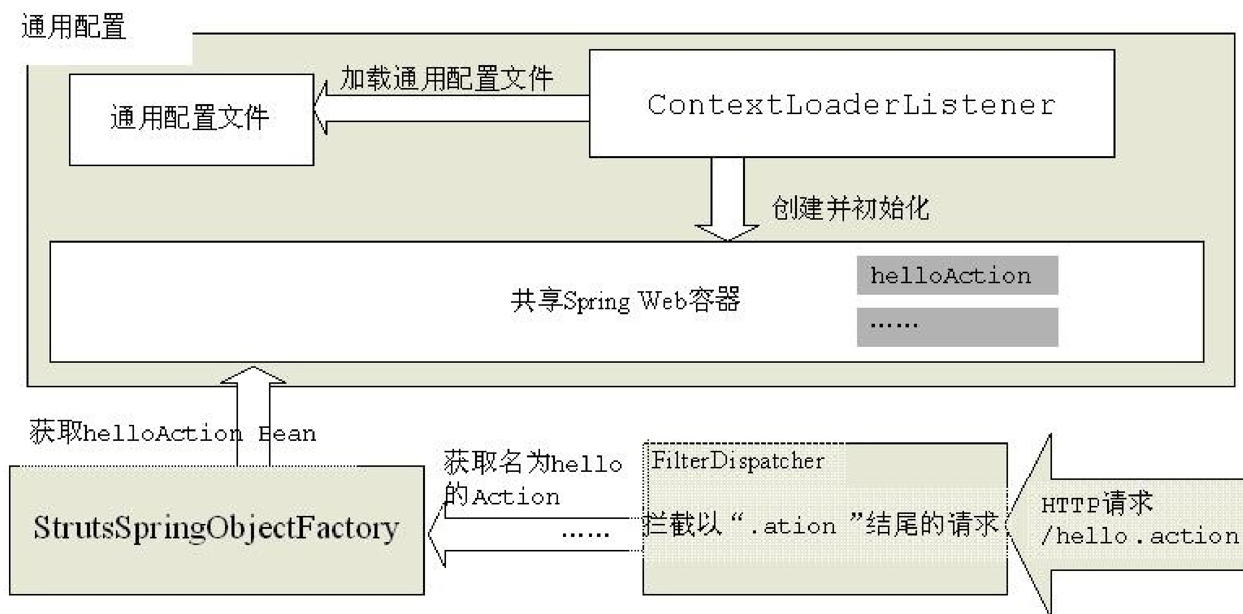


图10-5 Strut2与Spring集成

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2512.html】

【第十章】集成其它Web框架 之 10.4 集成JSF ——跟我学spring3

先进行通用配置，【第十章】集成其它Web框架 之 10.1 概述

10.4 集成JSF

10.4.1 概述

JSF（JavaServer Faces）框架是Java EE标准之一，是一个基于组件及事件驱动的Web框架，JSF只是一个标准（规范），目前有很多厂家实现，如Oracle的默认标准实现Mojarra、Apache的MyFaces、Jboss的RichFaces 等。

本示例将使用Oracle标准实现Mojarra，请到官网http://javaserverfaces.java.net/下载最新的JSF实现。

JSF目前有JSF1.1、JSF1.2、JSF2版本实现。

Spring集成JSF有三种种方式：

- 最简单集成：使用FacesContextUtils工具类的getWebApplicationContext方法，类似于Struts1x中的最简单实现；
- **VariableResolver**实现：Spring提供javax.faces.el.VariableResolver的两种实现DelegatingVariableResolver和SpringBeanVariableResolver，此方式适用于JSF1.1、JSF1.2及JSF2，但在JSF1.2和JSF2中不推荐使用该方式，而是使用第三种集成方式；
- **ELResolver**实现：Spring提供javax.el.ELResolver （Unified EL）实现SpringBeanFacesELResolver用于集成JSF1.2和JSF2。

接下来让我们首先让我们准备**JSF**所需要的**jar**包：

首先准备**JSF**所依赖的包：

- commons-digester.jar //必须，已有
- commons-collections.jar //必须，已有
- commons-beanutils.jar //必须，已有
- jsp-api.jar //必须，已有
- servlet-api.jar //必须，已有
- jstl.jar //可选
- standard.jar //可选
-

准备**JSF**包，到**http://javaserverfaces.java.net/**下载相应版本的**Mojarra**实现，如下载**JSF1.2**实现**mojarra-1.2_15-b01-FCS-binary.zip**，拷贝如下**jar**包到类路径：

- lib\jsf-api.jar //JSF规范接口包
- lib\jsf-impl.jar //JSF规范实现包

10.4.2 最简单集成

类似于Struts1x中的最简单集成，Spring集成JSF也提供类似的工具类FacesContextUtils，使用如下方式获取WebApplicationContext：

```
WebApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCu
```

当然我们不推荐这种方式，而是推荐使用接下来介绍的另外两种方式。

10.4.2 使用VariableResolver实现集成

Spring提供javax.faces.el.VariableResolver的两种实现DelegatingVariableResolver和SpringBeanVariableResolver，其都是Spring与JSF集成的中介者，此方式适用于JSF1.1、JSF1.2及JSF2：

- DelegatingVariableResolver：首先委托给JSF默认VariableResolver实现去查找JSF管理Bean，如果找不到再委托给Spring容器去查找Spring管理Bean；
- SpringBeanVariableResolver：其与DelegatingVariableResolver查找正好相反，首先委托给Spring容器去查找Spring管理Bean，如果找不到再委托给JSF默认VariableResolver实现去查找JSF管理Bean。

接下来看一下如何在JSF中集成Spring吧（本示例使用JSF1.2，其他版本的直接替换jar包即可）：

1、JSF管理Bean（Managed Bean）实现：

```
package cn.javass.spring.chapter10.jsf;
public class HelloBean {
    private String message;
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

2、JSF配置文件定义（resources/chapter10/jsf/faces-config.xml）：

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/w

    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
    </application>

    <managed-bean>
        <managed-bean-name>helloBean</managed-bean-name>
        <managed-bean-class>
            cn.javass.spring.chapter10.jsf.HelloBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>message</property-name>
            <value>#{message}</value>
        </managed-property>
    </managed-bean>
</faces-config>
```

- 与**Spring**集成：通过<variable-resolver>标签来指定集成Spring的中介者DelegatingVariableResolver；
- 注入**Spring**管理**Bean**：通过<managed-property>标签的**<value>#{message}</value>**注入Spring管理Bean"message"。

4、**JSP**页面定义（**webapp/hello-jsf.jsp**）：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    <h:outputText value="#{helloBean.message}"/>
</body>
</html>
</f:view>
```

5、**JSF**前端控制器定义，在**web.xml**中添加如下配置：

指定**JSF**配置文件位置，通过javax.faces.CONFIG_FILES上下文初始化参数指定JSF配置文件位置，多个可用"，"分割，如果不指定该参数则默认加载的配置文件为"/WEB-INF/ faces-config.xml"：

```
<!-- JSF配置文件开始 -->
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>
        /WEB-INF/classes/chapter10/jsf/faces-config-jsf1x.xml
    </param-value>
</context-param>
<!-- JSF配置文件结束 -->
```

前端控制器定义：使用FacesServlet作为JSF的前端控制器，其拦截以“.jsf”结尾的HTTP请求：

```
<!-- jsf前端控制器配置开始   -->
<servlet>
    <servlet-name>jsf</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>jsf</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<!-- jsf前端控制器配置结束   -->
```

7、执行测试，在**Web**浏览器中输入**http://localhost:8080/hello-jsf.jsp**可以看到**“Hello Spring”**信息说明**JSF**集成成功。

自此，JSF集成Spring已经成功，在此可以把DelegatingVariableResolver替换为SpringBeanVariableResolver，其只有在查找相应依赖时顺序是正好相反的，其他完全一样。

如果您的项目使用JSF1.2或JSF2，推荐使用SpringBeanFacesELResolver，因为其实标准的Unified EL实现，而且VariableResolver接口已经被注释为@Deprecated，表示可能在以后的版本中去掉该接口。

10.4.2 使用ELResolver实现集成

JSF1.2之前，JSP和JSF各个使用自己的一套表达式语言（EL Language），即如JSF使用VariableResolver实现来解析JSF EL表达式，而从JSF1.2和JSP2.1开始使用Unified EL，从而统一了表达式语言。

因此集成JSF1.2+可以通过实现Unified EL来完成集成，即Spring提供ELResolver接口实现SpringBeanFacesELResolver用于集成使用。

类似于VariableResolver实现，通过SpringBeanFacesELResolver集成首先将从Spring容器中查找相应的Spring管理Bean，如果没找到再通过默认的JSF ELResolver实现查找JSF管理Bean。

接下来看一下示例一下吧：

1、添加**Unified EL**所需要的**jar**包：

- el-api.jar //Unified EL规范接口包

由于在Jetty中已经包含了该api，因此该步骤可选。

2、修改**JSF**配置文件（**resources/chapter10/jsf/faces-config.xml**）：

将如下配置

```
<variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
</variable-resolver>
```

修改为：

```
<el-resolver>
org.springframework.web.jsf.el.SpringBeanFacesELResolver
</el-resolver>
```

3、执行测试，在**Web**浏览器中输入**http://localhost:8080/hello-jsf.jsp**可以看到**"Hello Spring"**信息说明**JSF**集成成功。

自此JSF与Spring集成就算结束了，是不是也很简单。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2513.html】

【第十一章】 SSH集成开发积分商城 之 11.1 概述 ——跟我学spring3

11.1 概述

11.1.1 功能概述

本节将通过介绍一个积分商城系统来演示如何使用SSH集成进行开发。

积分商城一般是购物网站的子模块，提供一些礼品或商品用于奖励老用户或使用积分来折换成现金，如图11-1所示。

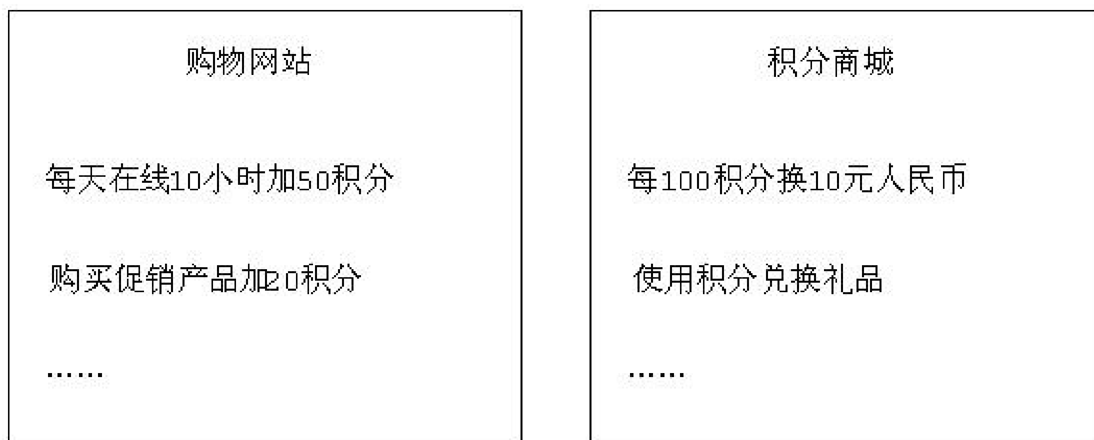


图11-1 购物网站与积分商城

积分商城功能点：

- 后台管理

交易管理模块：用于查看积分交易历史；

商品管理模块：用于CRUD积分兑换商品；

日报或月报：用于发送给运营人员每日积分兑换情况，一般通过email发送；

……

- 前台展示

商品展示：展示给用户可以使用积分兑换的商品；

支付模块：用户成功兑换商品后扣除用户相应积分

添加积分模块：提供接口用于其他产品赠送积分使用，如每天在线10小时赠送50积分，购买相应商品增加相应积分；

订单管理模块：订单管理模块可以使用现有购物平台的订单管理。

购物平台、用户系统及积分商城交互如图11-2所示，其中用户系统负责用户登录，购物平台是购物网站核心，积分商城用于用户使用积分购买商品。

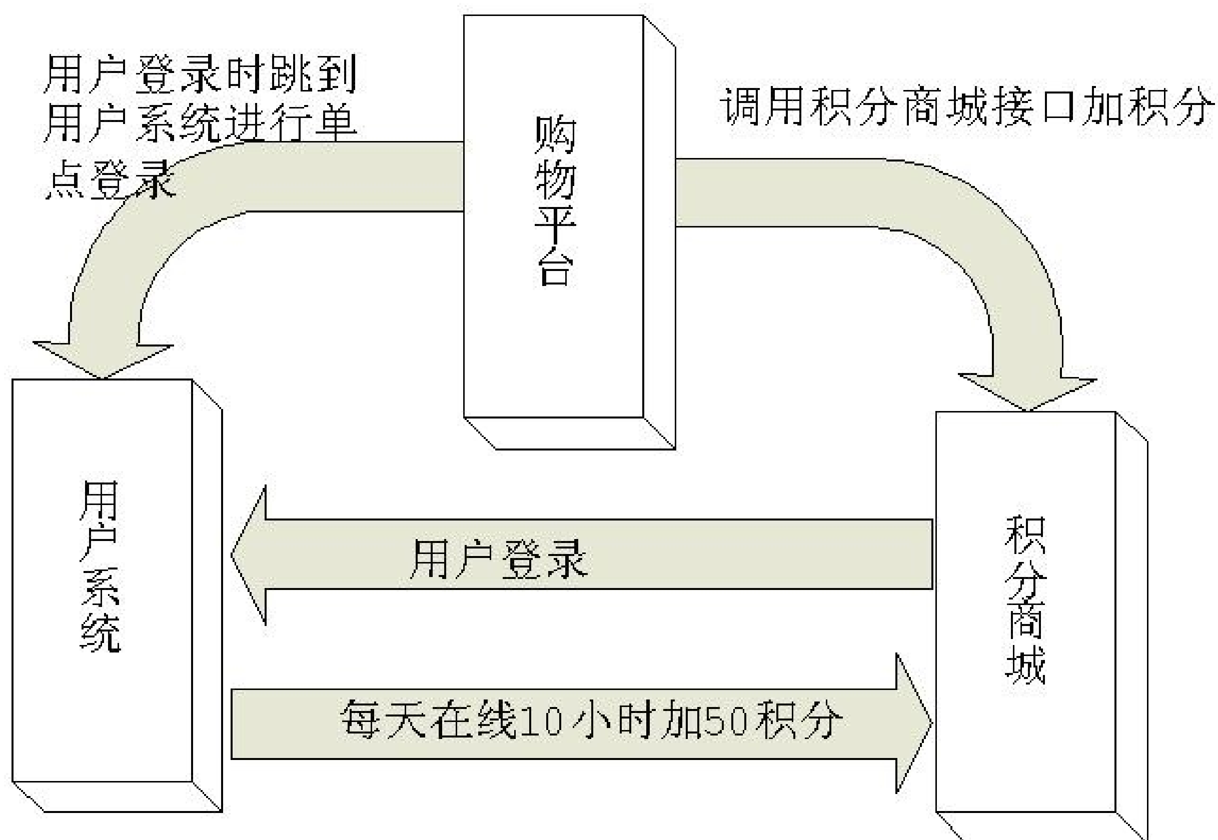


图11-2 购物平台、用户系统及积分商城交互

由于积分商城也是很复杂，由于篇幅原因不打算完全介绍，只介绍其中一个模块——商品（兑换码）管理及购买，该模块主要提供给用户使用积分兑换一些优惠券或虚拟物品（如移动充值卡）等等。

11.1.2技术选型

由于本节是关于SSH集成的，因此选用技术如下：

- 平台：Java EE；
- 运行环境：Windows XP，JDK1.6；
- 编辑器：Eclipse3.6 + SpringSource Tool Suite；
- Web容器：tomcat6.0.20；
- 数据库：mysql5.4.3；
- 框架：Struts2.0.14、Spring3.0.5、Hibernate3.6.0.Final；
- 日志记录：log4j1.2.15；
- 数据库连接池：proxool0.9.1；
- 视图技术：JSP 2.0。

技术选定了，应该考虑平台架构了，这关系到项目的成功与否。

11.1.3系统架构

积分商城系统架构也将采用经典的三层架构，如图11-3所示：



图11-3 三层架构

分层的目的是约束层次边界，每层的职责和目标应明确和单一，每层专注自己的事情，不要跨越分层边界，具体每层功能如下：

- 数据访问层：封装底层数据库或文件系统访问细节，从而对业务逻辑层提供一致的接口，使业务逻辑层不关心底层细节；
- 业务逻辑层：专注于业务逻辑实现，不关心底层如何访问，并在该层实现如声明式事务管理，组装分页对象；
- 表现层：应该非常轻量级及非常“薄（功能非常少，几乎全是委托）”，拦截用户请求并响应，表现层数据验证，负责根据请求委托给业务逻辑层进行业务处理，本层不实现任何业务逻辑，且提供用户交互界面；
- 数据模型层：数据模型定义，提供给各层使用，不应该算作三层架构中的某一层，因为数据模型可使用其他对象（如Map）代替之。

系统架构已选定，在此我们进行优化一下，因为在进行基于SSH的三层架构进行开发时通常会有一些通用功能、如通用DAO、通用Service、通用Action、通用翻页等等，因此我们再进行开发时都是基于通用功能进行的，能节省不少开发时间，从而可以使用这些节约的时间干自己想干的事情，如图10-4所示。

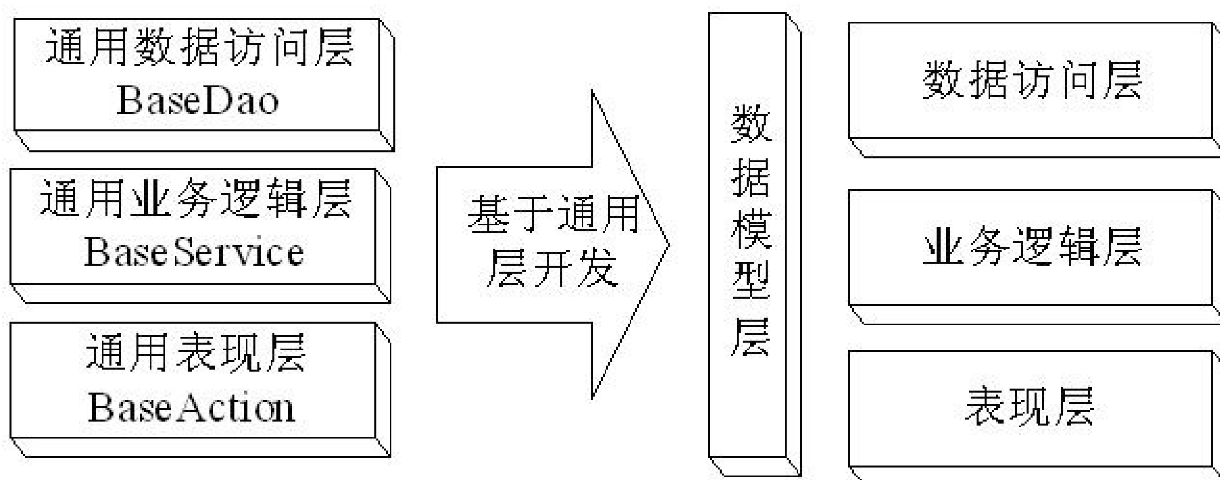


图10-4 基于通用层的三层架构

11.1.4项目搭建

1、创建动态**web**工程：

通过【File】>【New】>【other】>【Web】>【Dynamic Web Project】创建一个Web工程，如图11-5所示；

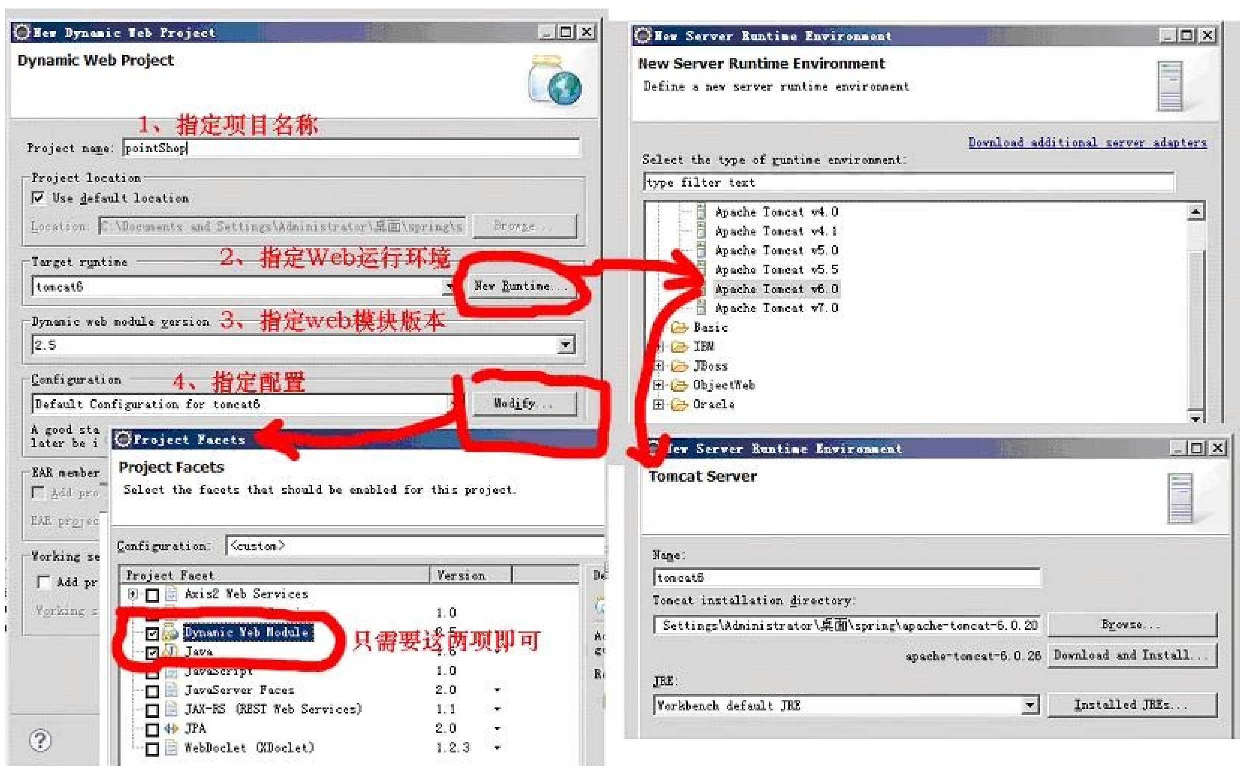
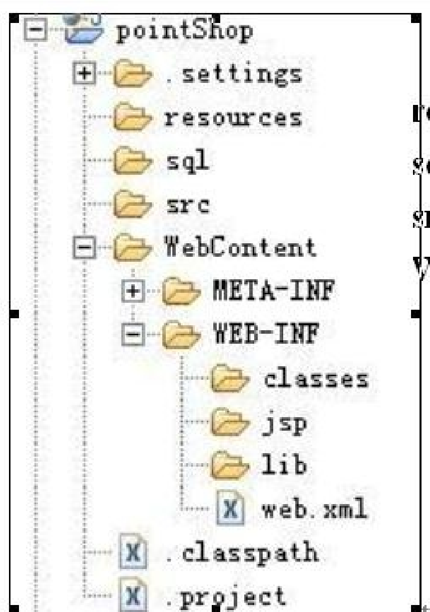


图11-5 Web工程配置

1、 项目结构，如图**11-6**所示：



resources：放置配置文件；

sql：放置 DDL 或 DML 语句，如数据库创建语句；

src：java 文件位置；

WebContent：web 项目根目录；

classes：存放编译好的 class 文件；

jsp：放置 jsp 视图文件；

lib 放置 jar 包；

web.xml：Web 项目部署描述符文件。

图11-6 项目结构

3、项目属性修改：

3.1、字符编码修改，如图11-7所示，在实际项目中一定要统一字符编码：

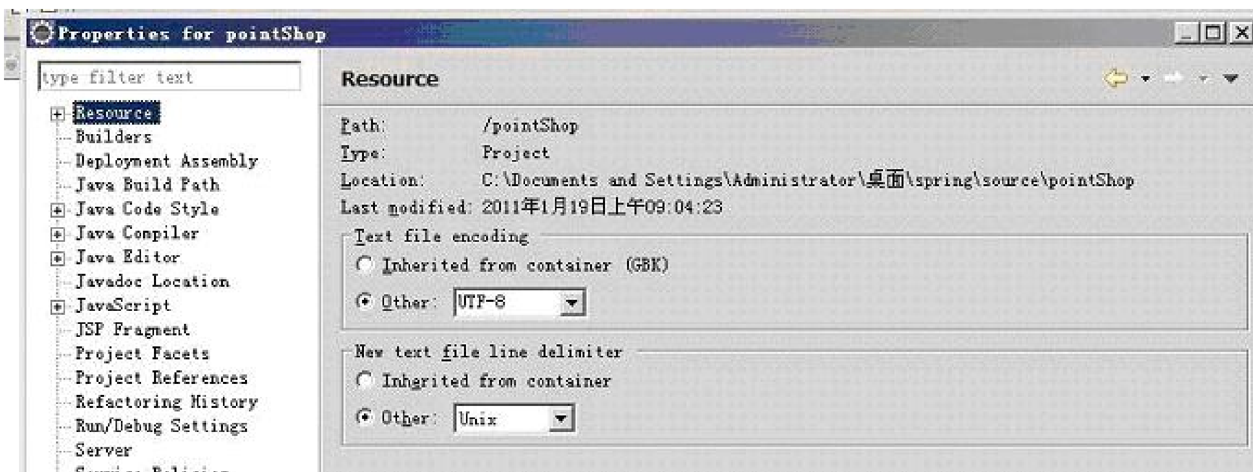


图11-7 修改项目字符编码

3.2、类路径输出修改，如图11-8，将类路径输出改为/WEB-INF/classes下：

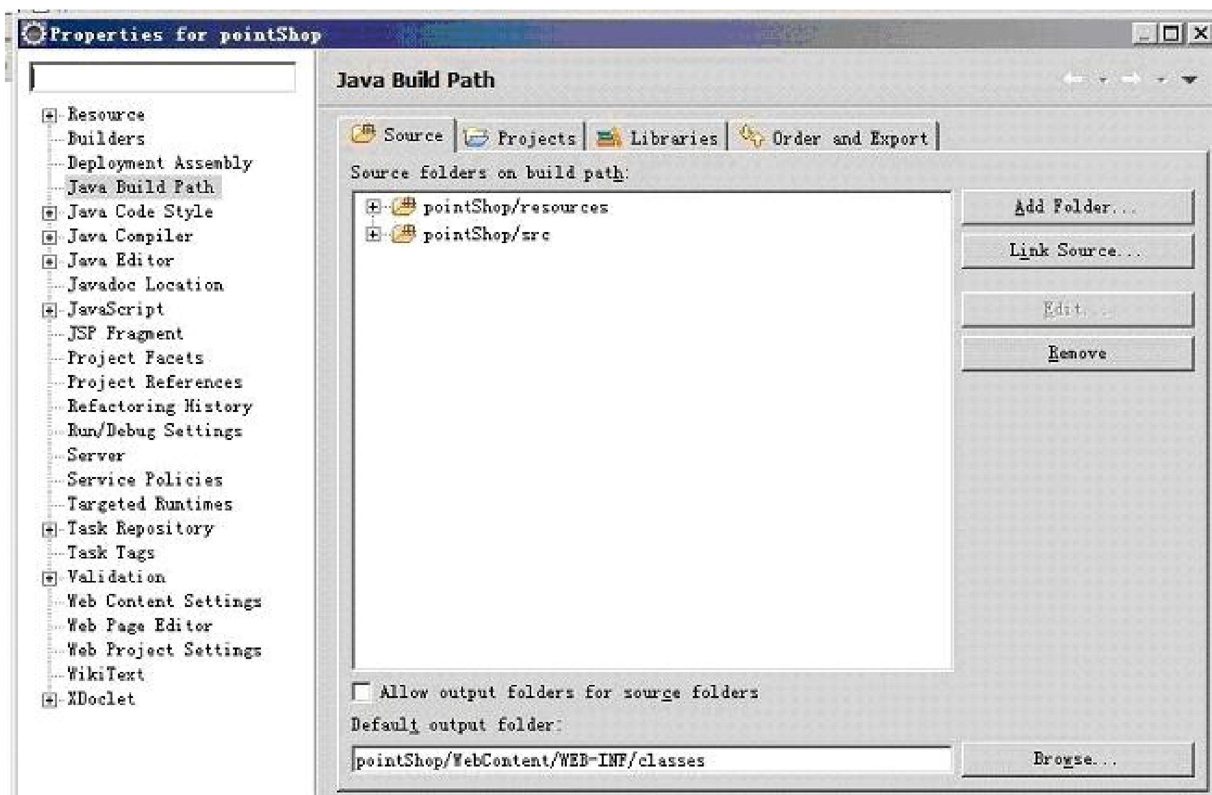


图11-8 类路径修改

4、准备**jar**包：

4.1、**Spring**项目依赖包，到下载的**spring-framework-3.0.5.RELEASE-with-docs.zip**中拷贝如下**jar**包：

- dist\org.springframework.aop-3.0.5.RELEASE.jar
- dist\org.springframework.asm-3.0.5.RELEASE.jar
- dist\org.springframework.beans-3.0.5.RELEASE.jar
- dist\org.springframework.context-3.0.5.RELEASE.jar
- dist\org.springframework.core-3.0.5.RELEASE.jar
- dist\org.springframework.expression-3.0.5.RELEASE.jar
- dist\org.springframework.jdbc-3.0.5.RELEASE.jar
- dist\org.springframework.orm-3.0.5.RELEASE.jar
- dist\org.springframework.transaction-3.0.5.RELEASE.jar
- dist\org.springframework.web-3.0.5.RELEASE.jar

4.2、**Spring**及其他项目依赖包，到**spring-framework-3.0.5.RELEASE-dependencies.zip**中拷贝如下**jar**吧：

- com.springsource.net.sf.cglib-2.2.0.jar
- com.springsource.org.aopalliance-1.0.0.jar
- com.springsource.org.apache.commons.beanutils-1.8.0.jar
- com.springsource.org.apache.commons.collections-3.2.1.jar
- com.springsource.org.apache.commons.digester-1.8.1.jar

- com.springsource.org.apache.commons.logging-1.1.1.jar
- com.springsource.org.apache.log4j-1.2.15.jar
- com.springsource.org.apache.taglibs.standard-1.1.2.jar
- com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

4.3、Hibernate依赖包，到hibernate-distribution-3.6.0.Final.zip中拷贝如下jar包：

- hibernate3.jar
- lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar
- lib\required\dom4j-1.6.1.jar
- lib\required\javassist-3.12.0.GA.jar
- lib\required\jta-1.1.jar
- lib\ required \slf4j-api-1.6.1.jar
- lib\required\antlr-2.7.6.jar

4.4、数据库连接池依赖包，到proxool-0.9.1.zip中拷贝如下jar包：

- lib\proxool-0.9.1.jar
- lib\proxool-cglib.jar

4.5、准备mysql JDBC连接依赖包：

- mysql-connector-java-5.1.10.jar

4.6、slf4j依赖包准备，到下载的slf4j-1.6.1.zip包中拷贝如下jar包：

- slf4j-log4j12-1.6.1.jar

4.7、Strut2依赖包，到struts-2.2.1.1.zip中拷贝如下jar包：

- lib\struts2-core-2.2.1.1.jar
- lib\xwork-core-2.2.1.1.jar
- lib\freemarker-2.3.16.jar
- lib\ognl-3.0.jar
- lib\struts2-spring-plugin-2.2.1.1.jar
- lib\commons-fileupload-1.2.1.jar

jar包终于准备完了，是不是很头疼啊，在此推荐使用maven进行依赖管理，无需拷贝这么多jar包，而是通过配置方式来指定使用的依赖，具体maven知识请到官方网站http://maven.apache.org/了解。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/2514.html】

【第十一章】 SSH集成开发积分商城 之 11.2 实现通用层 ——跟我学spring3

11.2 实现通用层

11.2.1 功能概述

通过抽象通用的功能，从而复用，减少重复工作：

- 对于一些通用的常量使用一个专门的常量类进行定义；
- 对于视图分页，也应该抽象出来，如JSP做出JSP标签；
- 通用的数据层代码，如通用的CRUD，减少重复劳动，节约时间；
- 通用的业务逻辑层代码，如通用的CRUD，减少重复劳动，节约时间；
- 通用的表现层代码，同样用于减少重复，并提供更好的代码结构规范。

11.2.2 通用的常量定义

目标：在一个常量类中定义通用的常量的好处是如果需要修改这些常量值只需在一个地方修改即可，变的地方只有一处而不是多处。

如默认分页大小如果在多处硬编码定义为10，突然发生变故需要将默认分页大小10为5，怎么办？如果当初我们提取出来放在一个通用的常量类中是不是只有一处变动。

```
package cn.javass.commons;
public class Constants {
    public static final int DEFAULT_PAGE_SIZE = 5;  //默认分页大小
    public static final String DEFAULT_PAGE_NAME = "page";
    public static final String CONTEXT_PATH = "ctx";
}
```

如上代码定义了通用的常量，如默认分页大小。

11.2.2通用分页功能

分页功能是项目中必不可少的功能，因此通用分页功能十分有必要，有了通用的分页功能，即有了规范，从而保证视图页面的干净并节约了开发时间。

1、 分页对象定义，用于存放是否有上一页或下一页、当前页记录、当前页码、分页上下文，该对象是分页中必不可少对象，一般在业务逻辑层组装**Page**对象，然后传送到表现层展示，然后通用的分页标签使用该对象来决定如何显示分页：

```
package cn.javass.commons.pagination;
import java.util.Collections;
import java.util.List;
public class Page<E> {/** 表示分页中的一页。*/
    private boolean hasPre; //是否有上一页
    private boolean hasNext; //是否有下一页
    private List<E> items; //当前页包含的记录列表
    private int index; //当前页页码(起始为1)
    //省略setter
    public int getIndex() {
        return this.index;
    }
    public boolean isHasPre() {
        return this.hasPre;
    }
    public boolean isHasNext() {
        return this.hasNext;
    }
    public List<E> getItems() {
        return this.items == null ? Collections.<E>emptyList() : this.items;
    }
}
```

2、 分页标签实现，将使用Page对象数据决定如何展示分页，如图11-9和11-10所示：



图11-9 11-10 通用分页标签实现

图11-9和11-10展示了两种分页展示策略，由于分页标签和集成SSH没多大关系且不是必须的并由于篇幅问题不再列出分页标签源代码，有兴趣的朋友请参考cn.javass.commons.pagination.NavigationTag类文件。代码下载地址

11.2.3 通用数据访问层

目标：通过抽象实现最基本的CURD操作，提高代码复用，可变部分按需实现。

1、通用数据访问层接口定义

```
package cn.javass.commons.dao;
import java.io.Serializable;
import java.util.List;
public interface IBaseDao<M extends Serializable, PK extends Serializable> {
    public void save(M model);// 保存模型对象
    public void saveOrUpdate(M model);// 保存或更新模型对象
    public void update(M model);// 更新模型对象
    public void merge(M model);// 合并模型对象状态到底层会话
    public void delete(PK id);// 根据主键删除模型对象
    public M get(PK id);// 根据主键获取模型对象
    public int countAll();//统计模型对象对应数据库表中的记录数
    public List<M> listAll();//查询所有模型对象
    public List<M> listAll(int pn, int pageSize);// 分页获取所有模型对象
}
```

通用DAO接口定义了如CRUD通用操作，而可变的（如查询所有已发布的接口，即有条件查询等）需要在相应DAO接口中定义，并通过泛型“M”指定数据模型类型和“PK”指定数据模型主键类型。

2、通用数据访问层DAO实现

此处使用Hibernate实现，即实现是可变的，对业务逻辑层只提供面向接口编程，从而隐藏数据访问层实现细节。

实现时首先通过反射获取数据模型类型信息，并根据这些信息获取Hibernate对应的数据模型的实体名，再根据实体名组装出通用的查询和统计记录的HQL，从而达到同样目的。

注意我们为什么把实现生成HQL时放到init方法中而不是构造器中呢？因为SessionFactory是通过setter注入，setter注入晚于构造器注入，因此在构造器中使用SessionFactory会是null，因此放到init方法中，并在Spring配置文件中指定初始化方法为init来完成生成HQL。

```
package cn.javass.commons.dao.hibernate;
//为节省篇幅省略import
public abstract class BaseHibernateDao<M extends Serializable, PK extends Serializable> e
    private Class<M> entityClass;
    private String HQL_LIST_ALL;
    private String HQL_COUNT_ALL;
    @SuppressWarnings("unchecked")
    public void init() {//通过初始化方法在依赖注入完毕时生成HQL
        //1、通过反射获取注解“M”（即模型对象）的类类型
        this.entityClass = (Class<M>) ((ParameterizedType) getClass().getGenericSuperclas
        //2、得到模型对象的实体名
        String entityName = getSessionFactory().getClassMetadata(this.entityClass).getEnt
        //3、根据实体名生成HQL
        HQL_LIST_ALL = "from " + entityName;
        HQL_COUNT_ALL = " select count(*) from " + entityName;
    }
    protected String getListAllHql() {//获取查询所有记录的HQL
        return HQL_LIST_ALL;
    }
    protected String getCountAllHql() {//获取统计所有记录的HQL
        return HQL_COUNT_ALL;
   }
   public void save(M model) {
        getHibernateTemplate().save(model);
   }
   public void saveOrUpdate(M model) {
        getHibernateTemplate().saveOrUpdate(model);
   }
   public void update(M model) {
        getHibernateTemplate().update(model);
   }

   public void merge(M model) {
        getHibernateTemplate().merge(model);
   }
   public void delete(PK id) {
        getHibernateTemplate().delete(this.get(id));
   }
   public M get(PK id) {
        return getHibernateTemplate().get(this.entityClass, id);
   }
   public int countAll() {
        Number total = unique(getCountAllHql());
        return total.intValue();
  }
   public List<M> listAll() {
        return list(getListAllHql());
```

```
    }
    public List<M> listAll(int pn, int pageSize) {
         return list(getListAllHql(), pn, pageSize);
    }
    protected <T> List<T> list(final String hql, final Object... paramlist) {
        return list(hql, -1, -1, paramlist);//查询所有记录
    }
     /** 通用列表查询,当pn<=-1 且 pageSize<=-1表示查询所有记录
      * @param <T> 模型类型
      * @param hql Hibernate查询语句
      * @param pn  页码 从1开始，
      * @param pageSize 每页记录数
      * @param paramlist 可变参数列表
      * @return 模型对象列表
      */
     @SuppressWarnings("unchecked")
    protected <T> List<T> list(final String hql, final int pn, final int pageSize, final O
     return getHibernateTemplate().
         executeFind(new HibernateCallback<List<T>>() {
             public List<T> doInHibernate(Session session)  throws HibernateException, SQL
                 Query query = session.createQuery(hql);
                 if (paramlist != null) {
                     for (int i = 0; i < paramlist.length; i++) {
                         query.setParameter(i, paramlist[i]);//设置占位符参数
                     }
                 }
                 if (pn > -1 && pageSize > -1) {//分页处理
                     query.setMaxResults(pageSize);//设置将获取的最大记录数
                     int start = PageUtil.getPageStart(pn, pageSize);
                     if(start != 0) {
                         query.setFirstResult(start);//设置记录开始位置
                     }
                 }
                 return query.list();
             }
         });
     }
    /**  根据查询条件返回唯一一条记录
      * @param <T> 返回类型
      * @param hql Hibernate查询语句
      * @param paramlist 参数列表
      * @return 返回唯一记录 */
     @SuppressWarnings("unchecked")
     protected <T> T unique(final String hql, final Object... paramlist) {
         return getHibernateTemplate().execute(new HibernateCallback<T>() {
             public T doInHibernate(Session session) throws HibernateException, SQLExcepti
                 Query query = session.createQuery(hql);
                 if (paramlist != null) {
                     for (int i = 0; i < paramlist.length; i++) {
                         query.setParameter(i, paramlist[i]);
                     }
                 }
                 return (T) query.setMaxResults(1).uniqueResult();
             }
         });
    }
    //省略部分可选的便利方法，想了解更多请参考源代码
}
```

通用DAO实现代码相当长，但麻烦一次，以后有了这套通用代码将会让工作很轻松，该通用DAO还有其他便利方法因为本示例不需要且由于篇幅原因没加上，请参考源代码。

11.2.4 通用业务逻辑层

目标：实现通用的业务逻辑操作，将常用操作封装提高复用，可变部分同样按需实现。

1、通用业务逻辑层接口定义

```
package cn.javass.commons.service;
//由于篇幅问题省略import
public interface IBaseService<M extends Serializable, PK extends Serializable> {
    public M save(M model); //保存模型对象
    public void saveOrUpdate(M model);// 保存或更新模型对象
    public void update(M model);// 更新模型对象
    public void merge(M model);// 合并模型对象状态
    public void delete(PK id);// 删除模型对象
    public M get(PK id);// 根据主键获取模型对象
    public int countAll();//统计模型对象对应数据库表中的记录数
    public List<M> listAll();//获取所有模型对象
    public Page<M> listAll(int pn);// 分页获取默认分页大小的所有模型对象
    public Page<M> listAll(int pn, int pageSize);// 分页获取所有模型对象
}
```

3、 通用业务逻辑层接口实现

通用业务逻辑层通过将通用的持久化操作委托给DAO层来实现通用的数据模型CRUD等操作。

通过通用的setDao方法注入通用DAO实现，在各Service实现时可以通过强制转型获取各转型后的DAO。

```
package cn.javass.commons.service.impl;
//由于篇幅问题省略import
public abstract class BaseServiceImpl<M extends Serializable, PK extends Serializable> im
    protected IBaseDao<M, PK> dao;
    public void setDao(IBaseDao<M, PK> dao) {//需要依赖注入
        this.dao = dao;
    }
    public IBaseDao<M, PK> getDao() {
        return this.dao;
    }
    public M save(M model) {
        getDao().save(model);
        return model;
    }
    public void merge(M model) {
        getDao().merge(model);
    }
    public void saveOrUpdate(M model) {
        getDao().saveOrUpdate(model);
    }
    public void update(M model) {
        getDao().update(model);
    }
    public void delete(PK id) {
        getDao().delete(id);
    }
    public void deleteObject(M model) {
        getDao().deleteObject(model);
    }
    public M get(PK id) {
        return getDao().get(id);
    }
    public int countAll() {
        return getDao().countAll();
    }
    public List<M> listAll() {
        return getDao().listAll();
    }
    public Page<M> listAll(int pn) {
        return this.listAll(pn, Constants.DEFAULT_PAGE_SIZE);
    }
    public Page<M> listAll(int pn, int pageSize) {
        Integer count = countAll();
        List<M> items = getDao().listAll(pn, pageSize);
        return PageUtil.getPage(count, pn, items, pageSize);
    }
}
```

11.2.6通用表现层

目标：规约化常见请求和响应操作，将常见的CURD规约化，采用规约编程提供开发效率，减少重复劳动。

Struts2常见规约编程：

- 通用字段驱动注入：如分页字段一般使用“pn”或“page”来指定当前分页页码参数名，通过Struts2的字段驱动注入实现分页页码获取的通用化；
- 通用**Result**：对于CURD操作完全可以提取公共的Result名字，然后在Strust2配置文件中进行规约配置；
- 数据模型属性名：在页面展示中，如新增和修改需要向值栈或请求中设置数据模型，在

此我们定义统一的数据模型名如“model”，这样在项目组中形成约定，大家只要按照约定来能提高开发效率；

- 分页对象属性名：与数据模型属性名同理，在此我们指定为“page”；
- 便利方法：如获取值栈、请求等可以提供公司内部需要的便利方法。

1、通用表现层**Action**实现：

```
package cn.javass.commons.web.action;
import cn.javass.commons.Constants;
//省略import
public class BaseAction extends ActionSupport {
    /** 通用Result */
    public static final String LIST = "list";
    public static final String REDIRECT = "redirect";
    public static final String ADD = "add";
    /** 模型对象属性名*/
    public static final String MODEL = "model";
    /** 列表模型对象属性名*/
    public static final String PAGE = Constants.DEFAULT_PAGE_NAME;
    public static final int DEFAULT_PAGE_SIZE = Constants.DEFAULT_PAGE_SIZE;
    private int pn = 1;  /** 页码，默认为1 */
    //省略pn的getter和setter，自己补上
    public ActionContext getActionContext() {
        return ActionContext.getContext();
    }
    public ValueStack getValueStack() {//获取值栈的便利方法
        return getActionContext().getValueStack();
    }
}
```

2、通用表现层**JSP**视图实现：

将视图展示的通用部分抽象出来减少页面设计的工作量。

2.1、通用**JSP**页头文件（**WEB-INF/jsp/common/inc/header.jsp**）：

此处实现比较简单，实际中可能包含如菜单等信息，对于可变部分使用请求参数来获取，从而保证了可变与不可变分离，如标题使用“${param.title}”来获取。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>${param.title}</title>
</head>
<body>
```

2.2、通用**JSP**页尾文件（**WEB-INF/jsp/common/inc/footer.jsp**）：

此处比较简单，实际中可能包含公司版权等信息。

```
</body>
</html>
```

2.3、通用**JSP**标签定义文件（**WEB-INF/jsp/common/inc/tld.jsp**）：

在一处定义所有标签，避免标签定义使代码变得凌乱，且如果有多个页面需要新增或删除标签即费事又费力。

```
<%@taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@taglib prefix="s" uri="/struts-tags" %
```

2.4、通用错误**JSP**文件（**WEB-INF/jsp/common/error.jsp**）：

当系统遇到错误或异常时应该跳到该页面来显示统一的错误信息并可能在该页保存异常信息。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<jsp:include page="inc/header.jsp"/>
失败或遇到异常！
<jsp:include page="inc/footer.jsp"/>
```

2.5、通用正确**JSP**文件（**WEB-INF/jsp/common/success.jsp**）：

对于执行成功的操作可以使用通用的页面表示，可变部分同样可以使用可变的请求参数传入。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<jsp:include page="inc/header.jsp"/>
成功！
<jsp:include page="inc/footer.jsp"/>
```

3、通用设置**web**环境上下文路径拦截器：

用于设置当前web项目的上下文路径，即可以在JSP页面使用“${ctx}”获取当前上下文路径。

```
package cn.javass.commons.web.filter;
//省略import
/** 用户设置当前web环境上下文，用于方便如JSP页面使用 */
public class ContextPathFilter implements Filter {
    @Override
    public void init(FilterConfig config) throws ServletException {
    }
    @Override
public void doFilter(
ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {
        String contextPath = ((HttpServletRequest) request).getContextPath();
        request.setAttribute(Constants.CONTEXT_PATH, contextPath);
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
}
```

11.2.7通用配置文件

目标：通用化某些常用且不可变的配置文件，同样目标是提高复用，减少工作量。

1、**Spring**资源配置文件（**resources/applicationContext-resources.xml**）：

定义如配置元数据替换Bean、数据源Bean等通用的Bean。

```
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="locations">
            <list>
                <value>classpath:resources.properties</value>
            </list>
        </property>
    </bean>
    <bean id="dataSource" class="org.springframework.jdbc.datasource.LazyConnectionDataSo
        <property name="targetDataSource">
            <bean class="org.logicalcobwebs.proxool.ProxoolDataSource">
                <property name="driver" value="${db.driver.class}" />
                <property name="driverUrl" value="${db.url}" />
                <property name="user" value="${db.username}" />
                <property name="password" value="${db.password}" />
                <property name="maximumConnectionCount" value="${proxool.maxConnCount}" /
                <property name="minimumConnectionCount" value="${proxool.minConnCount}" /
                <property name="statistics" value="${proxool.statistics}" />
                <property name="simultaneousBuildThrottle" value="${proxool.simultaneousB
                <property name="trace" value="${proxool.trace}" />
            </bean>
        </property>
    </bean>
</beans>
```

通过通用化如数据源来提高复用，对可变的如数据库驱动、URL、用户名等采用替换配置元数据形式进行配置，具体配置含义请参考【7.5集成Spring JDBC及最佳实践】。

2、替换配置元数据的资源文件（**resources/resources.properties**）：

定义替换配置元数据键值对用于替换Spring配置文件中可变的配置元数据。

```
#数据库连接池属性
proxool.maxConnCount=10
proxool.minConnCount=5
proxool.statistics=1m,15m,1h,1d
proxool.simultaneousBuildThrottle=30
proxool.trace=false
db.driver.class=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/point_shop?useUnicode=true&characterEncoding=utf8
db.username=root
db.password=
```

3、通用**Struts2**配置文件（**WEB-INF/struts.xml**）：

由于是要集成Spring，因此需要使用StrutsSpringObjectFactory，我们需要在action名字中出现“/”因此定义struts.enable.SlashesInActionNames=true。

在此还定义了“custom-default”包继承struts-default包，且是抽象的，在包里定义了如全局结果集全局异常映射。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.objectFactory" value="org.apache.struts2.spring.StrutsSpringOb
    <!-- 允许action的名字中出现"/" -->
    <constant name="struts.enable.SlashesInActionNames" value="true"/>
    <package name="custom-default" extends="struts-default" abstract="true">
        <global-results>
            <result name="success">/WEB-INF/jsp/common/success.jsp</result>
            <result name="error">/WEB-INF/jsp/common/error.jsp</result>
            <result name="exception">/WEB-INF/jsp/common/error.jsp</result>
        </global-results>
        <global-exception-mappings>
           <exception-mapping result="exception" exception="java.lang.Exception"/>
        </global-exception-mappings>
    </package>
</struts>
```

4、通用**log4j**日志记录配置文件（**resources/log4j.xml**）：

可以配置基本的log4j配置文件然后在其他地方通过拷贝来定制需要的日志记录配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
       <!-- Appenders -->
       <appender name="console" class="org.apache.log4j.ConsoleAppender">
              <param name="Target" value="System.out" />
              <layout class="org.apache.log4j.PatternLayout">
                     <param name="ConversionPattern" value="%-5p: %c - %m%n" />
              </layout>
       </appender>
       <!-- Root Logger -->
       <root>
              <priority value="DEBUG" />
              <appender-ref ref="console" />
       </root>
</log4j:configuration>
```

4、通用**web.xml**配置文件定义（**WEB-INF/web.xml**）：

定义如通用的集成配置、设置**web**环境上下文过滤器、字符过滤器（防止乱码）、通用的**Web**框架拦截器（如**Struts2**的）等等，从而可以通过拷贝复用。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
    <!-- 通用配置开始   -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:applicationContext-resources.xml
        </param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 通用配置结束   -->
    <!-- 设置web环境上下文（方便JSP页面获取）开始   -->
    <filter>
        <filter-name>Set Context Path</filter-name>
        <filter-class>cn.javass.commons.web.filter.ContextPathFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>Set Context Path</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!-- 设置web环境上下文（方便JSP页面获取）结束   -->
    <!-- 字符编码过滤器（防止乱码）开始   -->
    <filter>
        <filter-name>Set Character Encoding</filter-name>
        <filter-class>
            org.springframework.web.filter.CharacterEncodingFilter
        </filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>Set Character Encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!-- 字符编码过滤器（防止乱码）结束   -->
    <!-- Struts2.x前端控制器配置开始   -->
    <filter>
        <filter-name>struts2Filter</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2Filter</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- Struts2.x前端控制器配置结束   -->
</web-app>
```

【第十一章】SSH集成开发积分商城 之 11.3 实现积分商城层 ——跟我学spring3

11.3 实现积分商城层

11.3.1 概述

积分商城是基于通用层之上进行开发，这样我们能减少很多重复的劳动，加快项目开发进度。

11.3.2 实现数据模型层

1、商品表，定义了如商品名称、简介、原需积分、现需积分等，其中是否发布表示只有发布（true）了的商品才会在前台删除，是否已删除表示不会物理删除，商品不应该物理删除，而是逻辑删除，版本属性用于防止并发更新。

```
package cn.javass.point.model;
/** 商品表 */
@Entity
@Table(name = "tb_goods")
public class GoodsModel implements java.io.Serializable {
     /** 主键 */
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", length = 10)
    private int id;
    /** 商品名称 */
    @Column(name = "name", nullable = false, length = 100)
    private String name;
    /** 商品简介 */
    @Column(name = "description", nullable = false, length = 100)
    private String description;
    /** 原需积分 */
    @Column(name = "original_point", nullable = false, length = 10)
    private int originalPoint;
    /** 现需积分 */
    @Column(name = "now_point", nullable = false, length = 10)
    private int nowPoint;
    /** 是否发布，只有发布的在前台显示 */
    @Column(name = "published", nullable = false)
    private boolean published;
    /** 是否删除，商品不会被物理删除的 */
    @Column(name = "is_delete", nullable = false)
    private boolean deleted;
    /** 版本 */
    @Version  @Column(name = "version", nullable = false, length = 10)
    private int version;
    //省略getter和setter、hashCode及equals，实现请参考源代码
}
```

2、商品兑换码表，定义了兑换码、兑换码所属商品（兑换码和商品直接是多对一关系）、购买人、购买时间、是否已经购买（防止一个兑换码多个用户兑换）、版本。

```
package cn.javass.point.model;
import java.util.Date;
//省略部分import
/** 商品兑换码表 */
@Entity
@Table(name = "tb_goods_code")
public class GoodsCodeModel implements java.io.Serializable {
     /** 主键 */
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", length = 10)
    private int id;
    /** 所属商品 */
    @ManyToOne
    private GoodsModel goods;
    /** 兑换码*/
    @Column(name = "code", nullable = false, length = 100)
    private String code;
    /** 兑换人,实际环境中应该和用户表进行对应*/
    @Column(name = "username", nullable = true, length = 100)
    private String username;

    /** 兑换时间*/
    @Column(name = "exchange_time")
    private Date exchangeTime;
    /** 是否已经兑换*/
    @Column(name = "exchanged")
    private boolean exchanged = false;
    /** 版本 */
    @Version
    @Column(name = "version", nullable = false, length = 10)
    private int version;
    //省略getter和setter、hashCode及equals，实现请参考源代码
}
```

3、商品表及商品兑换码表之间关系，即一个商品有多个兑换码，如图11-10所示：

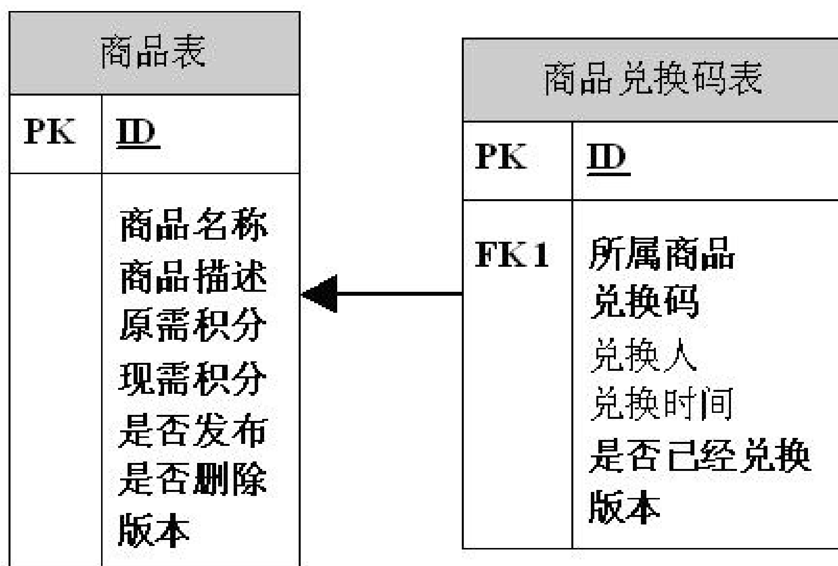


图11-10商品表及商品兑换码表之间关系

4、创建数据库及表结构的**SQL**语句文件（**sql/ pointShop_schema.sql**）：

```
CREATE DATABASE IF NOT EXISTS `point_shop`
DEFAULT CHARACTER SET 'utf8';
USE `point_shop`;
DROP TABLE IF EXISTS `tb_goods_code`;
DROP TABLE IF EXISTS `tb_goods`;
-- ----------------------------
-- Table structure for 商品表
-- ----------------------------
CREATE TABLE `tb_goods` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '商品id',
  `name` varchar(100) NOT NULL COMMENT '商品名称',
  `description` varchar(100) NOT NULL COMMENT '商品简介',

  `original_point` int(10) unsigned NOT NULL  COMMENT '原需积分',
  `now_point` int(10) unsigned NOT NULL  COMMENT '现需积分',
  `published` bool NOT NULL  COMMENT '是否发布',
  `is_delete` bool NOT NULL DEFAULT false COMMENT '是否删除',
  `version` int(10) unsigned NOT NULL DEFAULT 0 COMMENT '版本',
   PRIMARY KEY (`id`),
   INDEX(`name`),
   INDEX(`published`)
  )ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='商品表';
-- ----------------------------
-- Table structure for 商品兑换码表
-- ----------------------------
CREATE TABLE `tb_goods_code` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键id',
  `username` varchar(100) COMMENT '兑换用户',
  `goods_id` int(10) unsigned NOT NULL COMMENT '所属商品id',
  `code` varchar(100)  NOT NULL COMMENT '积分',
  `exchange_time` datetime  COMMENT '购买时间',
  `exchanged` bool DEFAULT false COMMENT '是否已经兑换',
  `version` int(10) unsigned NOT NULL DEFAULT 0 COMMENT '版本',
   PRIMARY KEY (`id`),
   FOREIGN KEY (`goods_id`) REFERENCES `tb_goods` (`id`)  ON DELETE CASCADE
  )ENGINE=InnoDB AUTO_INCREMENT=1000000 DEFAULT CHARSET=utf8 COMMENT='商品兑换码表';
```

Mysql数据库引擎应该使用InnoDB，如果使用MyISM将不支持事务。

11.3.3 实现数据访问层

数据访问层只涉及与底层数据库或文件系统等打交道，不会涉及业务逻辑，一定注意层次边界，不要在数据访问层实现业务逻辑。

商品模块的应该实现如下功能：

- 继承通用数据访问层的CRUD功能；
- 分页查询所有已发布的商品
- 统计所有已发布的商品；

商品兑换码模块的应该实现如下功能：

- 继承通用数据访问层的CRUD功能；
- 根据商品ID分页查询该商品的兑换码
- 根据商品ID统计该商品的兑换码记录数；
- 根据商品ID获取一个还没有兑换的商品兑换码

1、商品及商品兑换码**DAO**接口定义：

商品及商品兑换码DAO接口定义直接继承IBaseDao，无需在这些接口中定义重复的CRUD方法了，并通过泛型指定数据模型类及主键类型。

```
package cn.javass.point.dao;
//省略import
/** 商品模型对象的DAO接口 */
public interface IGoodsDao extends IBaseDao<GoodsModel, Integer> {
    /** 分页查询所有已发布的商品*/
    List<GoodsModel> listAllPublished(int pn);
    /** 统计所有已发布的商品记录数*/
    int countAllPublished();
}
```

```
package cn.javass.point.dao;
//省略import
/** 商品兑换码模型对象的DAO接口 */
public interface IGoodsCodeDao extends IBaseDao<GoodsCodeModel, Integer> {
    /** 根据商品ID统计该商品的兑换码记录数*/
    public int countAllByGoods(int goodsId);
    /** 根据商品ID查询该商品的兑换码列表*/
    public List<GoodsCodeModel> listAllByGoods(int pn, int goodsId);
    /** 根据商品ID获取一个还没有兑换的商品兑换码 */
    public GoodsCodeModel getOneNotExchanged(int goodsId);
}
```

2、 商品及商品兑换码**DAO**接口实现定义：

DAO接口实现定义都非常简单，对于CRUD实现直接从BaseHibernateDao继承即可，无需再定义重复的CRUD实现了，并通过泛型指定数据模型类及主键类型。

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsHibernateDao extends BaseHibernateDao<GoodsModel, Integer> implements I
    @Override  //覆盖掉父类的delete方法，不进行物理删除
    public void delete(Integer id) {
        GoodsModel goods = get(id);
        goods.setDeleted(true);
        update(goods);
    }
    @Override //覆盖掉父类的getCountAllHql方法，查询不包括逻辑删除的记录
    protected String getCountAllHql() {
        return super.getCountAllHql() + " where deleted=false";
    }
    @Override //覆盖掉父类的getListAllHql方法，查询不包括逻辑删除的记录
    protected String getListAllHql() {
        return super.getListAllHql() + " where deleted=false";
    }
    @Override //统计没有被逻辑删除的且发布的商品数量
    public int countAllPublished() {
        String hql = getCountAllHql() + " and published=true";
        Number result = unique(hql);
            return result.intValue();
    }
    @Override //查询没有被逻辑删除的且发布的商品
    public List<GoodsModel> listAllPublished(int pn) {
        String hql = getListAllHql() + " and published=true";
        return list(hql, pn, Constants.DEFAULT_PAGE_SIZE);
    }
}
```

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsCodeHibernateDao extends
BaseHibernateDao<GoodsCodeModel, Integer> implements IGoodsCodeDao {
@Override  //根据商品ID查询该商品的兑换码
    public List<GoodsCodeModel> listAllByGoods(int pn, int goodsId) {
        final String hql = getListAllHql() + " where goods.id = ?";
        return list(hql, pn, Constants.DEFAULT_PAGE_SIZE , goodsId);
    }
@Override //根据商品ID统计该商品的兑换码数量
    public int countAllByGoods(int goodsId) {
        final String hql = getCountAllHql() + " where goods.id = ?";
        Number result = unique(hql, goodsId);
        return result.intValue();
    }
```

3、**Spring DAO**层配置文件（**resources/cn/javass/point/dao/ applicationContext-hibernate.xml**）：

DAO配置文件中定义Hibernate的SessionFactory、事务管理器和DAO实现。

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.Annotation
    <property name="dataSource" ref="dataSource"/><!-- 1、指定数据源 -->
    <property name="annotatedClasses">           <!-- 2、指定注解类 -->
        <list>
             <value>cn.javass.point.model.GoodsModel</value>
             <value>cn.javass.point.model.GoodsCodeModel</value>
        </list>
    </property>
    <property name="hibernateProperties"><!-- 3、指定Hibernate属性 -->
       <props>
           <prop key="hibernate.dialect">${hibernate.dialect}</prop>
           <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
           <prop key="hibernate.format_sql">${hibernate.format_sql}</prop>
           <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
        </props>
    </property>
</bean>
<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManage
      <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

```
<bean id="abstractDao" abstract="true" init-method="init">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="goodsDao" class="cn.javass.point.dao.hibernate.GoodsHibernateDao" parent="abstr
<bean id="goodsCodeDao" class="cn.javass.point.dao.hibernate.GoodsCodeHibernateDao" paren
```

4、修改替换配置元数据的资源文件（**resources/resources.properties**），添加**Hibernate**属性相关：

```
#Hibernate属性
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
hibernate.hbm2ddl.auto=none
hibernate.show_sql=false
hibernate.format_sql=true
```

11.3.4 实现业务逻辑层

业务逻辑层实现业务逻辑，即系统中最复杂、最核心的功能，不应该在业务逻辑层出现如数据库访问等底层代码，对于这些操作应委托给数据访问层实现，从而保证业务逻辑层的独立性和可复用性，并应该在业务逻辑层组装分页对象。

商品模块应该实现如下功能：

- CURD操作，直接委托给通用业务逻辑层；
- 根据页码查询所有已发布的商品的分页对象，即查询指定页的记录，这是和数据访问层不同的；

商品兑换码模块应该实现如下功能：

- CURD操作，直接委托给通用业务逻辑层；
- 根据页码和商品Id查询查询所有商品兑换码分页对象，即查询指定页的记录；
- 新增指定商品的兑换码，用于对指定商品添加兑换码；
- 购买指定商品兑换码操作，用户根据商品购买该商品的兑换码，如果指定商品的兑换码没有了将抛出没有兑换码异常NotCodeException；

1、商品及商品兑换码**Service**接口定义：

接口定义时，对于CRUD直接继承IBaseService即可，无需再在这些接口中定义重复的CRUD方法了，并通过泛型指定数据模型类及数据模型的主键。

```
package cn.javass.point.service;
//省略import
public interface IGoodsService extends IBaseService<GoodsModel, Integer> {
    /**根据页码查询所有已发布的商品的分页对象*/
    Page<GoodsModel> listAllPublished(int pn);
}
```

```
package cn.javass.point.service;
//省略import
public interface IGoodsCodeService extends IBaseService<GoodsCodeModel, Integer> {
    /** 根据页码和商品Id查询查询所有商品兑换码分页对象*/
    public Page<GoodsCodeModel> listAllByGoods(int pn, int goodsId);
    /** 新增指定商品的兑换码*/
    public void save(int goodsId, String[] codes);
    /** 购买指定商品兑换码 */
    GoodsCodeModel buy(String username, int goodsId) throws NotCodeException ;
}
```

2、**NotCodeException**异常定义，表示指定商品的兑换码已经全部被兑换了，没有剩余的兑换码了：

```
package cn.javass.point.exception;
/** 购买失败异常,表示没有足够的兑换码 */
public class NotCodeException extends RuntimeException {
}
```

NotCodeException异常类实现RuntimeException，当需要更多信息时可以在异常中定义，异常比硬编码错误代码（如-1表示没有足够的兑换码）更好理解。

3、商品及商品兑换码**Service**接口实现定义：

接口实现时，CRUD实现直接从BaseServcice继承即可，无需再在这些专有实现中定义重复的CRUD实现了，并通过泛型指定数据模型类及数据模型的主键。

```
package cn.javass.point.service.impl;
//省略import
public class GoodsServiceImpl extends BaseServiceImpl<GoodsModel, Integer> implements IGo
    @Override
    public Page<GoodsModel> listAllPublished(int pn) {
        int count = getGoodsDao().countAllPublished();
        List<GoodsModel> items = getGoodsDao().listAllPublished(pn);
        return PageUtil.getPage(count, pn, items, Constants.DEFAULT_PAGE_SIZE);
    }
    IGoodsDao getGoodsDao() {//将通用DAO转型
        return (IGoodsDao) getDao();
    }
}
```

```
package cn.javass.point.service.impl;
//省略import
public class GoodsCodeServiceImpl extends BaseServiceImpl<GoodsCodeModel, Integer> implem
    private IGoodsService goodsService;
    public void setGoodsService(IGoodsService goodsService) {//注入IGoodsService
        this.goodsService = goodsService;
    }
    private IGoodsCodeDao getGoodsCodeDao() {//将注入的通用DAO转型
        return (IGoodsCodeDao) getDao();
    }
    @Override
    public Page<GoodsCodeModel> listAllByGoods(int pn, int goodsId) {
        Integer count = getGoodsCodeDao().countAllByGoods(goodsId);
        List<GoodsCodeModel> items = getGoodsCodeDao().listAllByGoods(pn, goodsId);
        return PageUtil.getPage(count, pn, items, Constants.DEFAULT_PAGE_SIZE);
    }
    @Override
    public void save(int goodsId, String[] codes) {
        GoodsModel goods = goodsService.get(goodsId);
        for(String code : codes) {
            if(StringUtils.hasText(code)) {
                GoodsCodeModel goodsCode = new GoodsCodeModel();
                goodsCode.setCode(code);
                goodsCode.setGoods(goods);
            save(goodsCode);
            }
        }
    }
    @Override
    public GoodsCodeModel buy(String username, int goodsId)  throws NotCodeException {
        //1、实际实现时要验证用户积分是否充足
        //2、其他逻辑判断
        //3、实际实现时要记录交易记录开始
       GoodsCodeModel goodsCode = getGoodsCodeDao().getOneNotExchanged(goodsId);

       if(goodsCode == null) {
            //3、实际实现时要记录交易记录失败
            throw new NotCodeException();
            //目前只抛出一个异常，还可能比如并发购买情况
       }
       goodsCode.setExchanged(true);
       goodsCode.setExchangeTime(new Date());
       goodsCode.setUsername(username);
       save(goodsCode);
       //3、实际实现时要记录交易记录成功
       return goodsCode;
    }
}
```

save方法和buy方法实现并不是最优的，save方法中如果兑换码有上千个怎么办？这时就需要批处理了，通过批处理比如20条一提交数据库来提高性能。buy方法就要考虑多个用户同时购买同一个兑换码如何处理？

交易历史一定要记录，从交易开始到交易结束（不管成功与否）一定要记录用于当客户投诉时查询相应数据。

4、Spring Service层配置文件（resources/cn/javass/point/service/ applicationContext-service.xml）：

Service层配置文件定义了事务和Service实现。

```
<tx:advice id="txAdvice" transaction-manager="txManager">
<tx:attributes>
<tx:method name="save*" propagation="REQUIRED" />
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="create*" propagation="REQUIRED" />
        <tx:method name="insert*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="buy*" propagation="REQUIRED" />
        <tx:method name="count*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="list*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="*" propagation="SUPPORTS" read-only="true" />
    </tx:attributes>
</tx:advice>
```

```
<aop:config>
    <aop:pointcut id="txPointcut" expression="execution(* cn.javass.point.service.*.*(..)
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
<bean id="goodsService" class="cn.javass.point.service.impl.GoodsServiceImpl">
    <property name="dao" ref="goodsDao"/>
</bean>
<bean id="goodsCodeService" class="cn.javass.point.service.impl.GoodsCodeServiceImpl">
    <property name="dao" ref="goodsCodeDao"/>
    <property name="goodsService" ref="goodsService"/>
</bean>
```

11.3.5 实现表现层

表现层显示页面展示和交互，应该支持多种视图技术（如JSP、Velocity），表现层实现不应该实现诸如业务逻辑层功能，只负责调用业务逻辑层查找数据模型并委托给相应的视图进行展示数据模型。

积分商城分为前台和后台，前台负责与客户进行交互，如购买商品；后台是负责商品及商品兑换码维护的，只应该管理员有权限操作。

后台模块：

- 商品管理模块：负责商品的维护，包括列表、新增、修改、删除、查询所有商品兑换码功能；
- 商品兑换码管理模块：包括列表、新增、删除所有兑换码操作；

前台模块：只有已发布商品展示，用户购买指定商品时，如果购买成功则给用户发送兑换码，购买失败给用户错误提示。

表现层Action实现时一般使用如下规约编程：

- **Action**方法定义：使用如list方法表示展示列表，doAdd方法表示去新增页面，add方法表示提交新增页面的结果并委托给Service层进行处理；
- 结果定义：如使用“list”结果表示到展示列表页面，“add”结果去新增页面等等；

- 参数设置：一般使用如“model”表示数据模型，使用“page”表示分页对象。

1、集成**Struts2**和**Spring**配置：

1.1、**Spring Action**配置文件：即**Action**将从**Spring**容器中获取，前台和后台配置文件应该分开以便好管理；

- 后台Action配置文件resources/cn/javass/web/pointShop-admin-servlet.xml；
- 前台Action配置文件resources/cn/javass/web/pointShop-front-servlet.xml；

1.2、**Struts**配置文件定义（**resources/struts.xml**）：

为了提高开发效率和采用规约编程，我们将使用模式匹配通配符来定义action。对于管理后台和前台应该分开，URL模式将类似于/{module}/{action}/{method}.action：

- module即模块名如admin，action即action前缀名，如后台的“GoodsAction”可以使用“goods”，method即Action中的方法名如“list”。
- 可以在Struts配置文件中使用{1}访问第一个通配符匹配的结果，以此类推；
- Reuslt也采用规约编程，即只有符合规律的放置jsp文件才会匹配到，如Result为“/WEB-INF/jsp/admin/{1}/list.jsp”，而URL为/goods/list.action 结果将为“/WEB-INF/jsp/admin/goods/list.jsp”。

```
<package name="admin" extends="custom-default" namespace="/admin">
        <action name="*/*" class="/admin/{1}Action" method="{2}">
           <result name="redirect" type="redirect">/admin/{1}/list.action</result>
           <result name="list">/WEB-INF/jsp/admin/{1}/list.jsp</result>
           <result name="add">/WEB-INF/jsp/admin/{1}/add.jsp</result>
        </action>
    </package>
```

在此我们继承了“**custom-default**”包来支持action名字中允许“/”。

如“/admin/goods/list.action”将调用cn.javass.point.web.admin.action.GoodsAction的list方法。

```
<package name="front" extends="custom-default">
        <action name="*/*" class="/front/{1}Action" method="{2}">
           <result name="redirect" type="redirect">/{1}/list.action</result>
           <result name="list">/WEB-INF/jsp/front/{1}/list.jsp</result>
           <result name="add">/WEB-INF/jsp/front/{1}/add.jsp</result>
           <result name="buyResult">/WEB-INF/jsp/front/{1}/buyResult.jsp</result>
        </action>
    </package>
```

如“/goods/list.action”将调用cn.javass.point.web.front.action.GoodsAction的list方法。

1.3、**web.xml**配置：将**Spring**配置文件加上；

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:applicationContext-resources.xml,
        classpath:cn/javass/point/dao/applicationContext-hibernate.xml,
        classpath:cn/javass/point/service/applicationContext-service.xml,
        classpath:cn/javass/point/web/pointShop-admin-servlet.xml,
        classpath:cn/javass/point/web/pointShop-front-servlet.xml
    </param-value>
</context-param>
```

2、后台商品管理模块

商品管理模块实现商品的CRUD，本示例只演示新增，删除和更新由于篇幅问题留作练习。

2.1、Action实现

```
package cn.javass.point.web.admin.action;
//省略import
public class GoodsAction extends BaseAction {
    public String list() {//列表、展示所有商品（包括未发布的）
        getValueStack().set(PAGE, goodsService.listAll(getPn()));
        return LIST;
    }
    public String doAdd() {//到新增页面
        goods = new GoodsModel();
        getValueStack().set(MODEL, goods);
        return ADD;
    }
    public String add() {//保存新增模型对象
        goodsService.save(goods);
        return REDIRECT;
    }
    //字段驱动数据填充
    private int id = -1;    //前台提交的商品ID
    private GoodsModel goods; //前台提交的商品模型对象
    //省略字段驱动数据的getter和setter
    //依赖注入Service
    private IGoodsService goodsService;
    //省略依赖注入的getter和setter
}
```

2.2、Spring配置文件定义（resources/cn/javass/web/pointShop-admin-servlet.xml）：

```
<bean name="/admin/goodsAction" class="cn.javass.point.web.admin.action.GoodsAction" scop
    <property name="goodsService" ref="goodsService"/>
</bean>
```

2.3、JSP实现商品列表页面（WEB-INF/jsp/admin/goods/list.jsp）

查询所有商品，通过迭代“page.items”（Page对象的items属性中存放着分页列表数据）来显示商品列表，在最后应该有分页标签（请参考源代码，示例无），如类似于“<my:page url="${ctx}/admin/goods/list.action"/>”来定义分页元素。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-商品列表"/>
</jsp:include>
<a href="${ctx}/admin/goods/doAdd.action">新增</a><br/>
<table border="1">
    <tr>
        <th>ID</th>
        <th>商品名称</th>
        <th>商品描述</th>
        <th>原需积分</th>
        <th>现需积分</th>
        <th>是否已发布</th>
        <th></th>
        <th></th>
        <th></th>
</tr>
  <s:iterator value="page.items">
<tr>
        <td><a href="${ctx}/admin/goods/toUpdate.action?id=<s:property value='id'/>"><s:p
        <td><s:property value="name"/></td>
        <td><s:property value="description"/></td>
        <td><s:property value="originalPoint"/></td>
        <td><s:property value="nowPoint"/></td>
        <td><s:property value="published"/></td>
        <td>更新</td> <td>删除</td>
        <td><a href="${ctx}/admin/goodsCode/list.action?goodsId=<s:property value='id'/>"
    </tr>
    </s:iterator>
</table>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”项目选择【Run As】>【Run On Server】启动Tomcat服务器，在浏览器中输入“http://localhost:8080/pointShop/admin/goods/list.action”将显示图11-11界面。

新增

ID	商品名称	商品描述	原需积分	现需积分	是否已发布			
6	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码
7	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码
8	研磨设计模式	一本值得反复阅读的书	200	100	true	更新	删除	查看兑换码

图11-11 后台商品列表页面

2.4、JSP实现商品新增页面（WEB-INF/jsp/admin/goods/add.jsp）

表单提交到/admin/goods/add.action即cn.javass.point.web.admin.action.GoodsAction的add方法。并将参数绑定到goods属性上，在此我们没有进行数据验证，在实际项目中页面中和Action中都要进行数据验证。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-新增"/>
</jsp:include>
<s:fielderror cssStyle="color:red"/>
<s:form action="/admin/goods/add.action" method="POST" acceptcharset="UTF-8" >
<s:token/>
<table border="1">
    <s:hidden name="goods.id" value="%{model.id}"/>
    <s:hidden name="goods.version" value="%{model.version}"/>
    <tr>
        <s:textfield label="商品名称" name="goods.name" value="%{model.name}" required="tr
    </tr>

    <tr>
        <s:textarea label="商品简介" name="goods.description" value="%{model.description}"
    </tr>
    <tr>
        <s:textfield label="原需积分" name="goods.originalPoint" value="%{model.originalPo
    </tr>
    <tr>
        <s:textfield label="现需积分" name="goods.nowPoint" value="%{model.nowPoint}" requ
    </tr>
    <tr>
        <s:radio label="是否发布" name="goods.published" list="#{true:'发布',false:'不发布'}
    </tr>
    <tr>
      <td><input name="submit" type="submit" value="新增"/></td>
    <td>
        <input name="cancel" type="button" onclick="javascript:window.location.href='${ct
    </td>
    </tr>
</table>
</s:form>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Tomcat服务器，在商品列表页面单间【新增】按钮将显示图11-11界面。

商品名称*:	研磨设计模式
商品简介*:	一本值得反复阅读的书
原需积分*:	0
现需积分*:	0
是否发布:	发布 不发布
新增	取消

图11-12 后台商品新增页面

3、后台兑换码管理

提供根据商品ID查询兑换码列表及新增兑换码操作，兑换码通过文本框输入多个，使用换行分割。

3.1、Action实现

```
package cn.javass.point.web.admin.action;
//省略import
public class GoodsCodeAction extends BaseAction {
    public String list() {
        getValueStack().set(MODEL, goodsService.get(goodsId));
        getValueStack().set(PAGE,
       goodsCodeService.listAllByGoods(getPn(), goodsId));
        return LIST;
    }
    public String doAdd() {
        getValueStack().set(MODEL, goodsService.get(goodsId));
        return ADD;
    }
    public String add() {
        String[] codes = splitCodes();
        goodsCodeService.save(goodsId, codes);
        return list();
    }
    private String[] splitCodes() {//将根据换行分割code码
        if(codes == null) {
            return new String[0];
        }
        return codes.split("\r"); //简单起见不考虑“\n”
    }
    //字段驱动数据填充
    private int id = -1;      //前台提交的商品兑换码ID
    private int goodsId = -1; //前台提交的商品ID
    private String codes;//前台提交的兑换码，由换行分割
    private GoodsCodeModel goodsCode; //前台提交的商品兑换码模型对象
    //省略字段驱动数据的getter和setter
    //依赖注入Service
    private IGoodsCodeService goodsCodeService;
    private IGoodsService goodsService;
    //省略依赖注入的getter和setter
}
```

3.2、Spring配置文件定义（resources/cn/javass/web/pointShop-admin-servlet.xml）：

```
<bean name="/admin/goodsCodeAction"
class="cn.javass.point.web.admin.action.GoodsCodeAction" scope="prototype">
<property name="goodsService" ref="goodsService"/>
    <property name="goodsCodeService" ref="goodsCodeService"/>
</bean>
```

3.3、JSP实现商品兑换码列表页面（WEB-INF/jsp/admin/goodsCode/list.jsp）

商品兑换码列表页面时将展示相应商品的兑换码。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-商品Code码列表"/>
</jsp:include>
<a href="${ctx}/admin/goodsCode/doAdd.action?goodsId=${model.id}">新增</a>|
<a href="${ctx}/admin/goods/list.action">返回商品列表</a><br/>
<table border="1">
    <tr>
        <th>ID</th>
        <th>所属商品</th>
        <th>兑换码</th>
        <th>购买人</th>
        <th>兑换时间</th>
        <th>是否已经兑换</th>
        <th></th>
    </tr>
    <s:iterator value="page.items">
    <tr>
        <td><s:property value="id"/></td>
        <td><s:property value="goods.name"/></td>
        <td><s:property value="code"/></td>
        <td><s:property value="username"/></td>
        <td><s:date name="exchangeTime" format="yyyy-MM-dd"/></td>
        <td><s:property value="exchanged"/></td>
        <td>删除</td>
    </tr>
    </s:iterator>
</table>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Web服务器，在浏览器中输入“http://localhost:8080/pointShop/admin/goods/list.action”，然后在指定商品后边点击【查看兑换码】将显示图11-15界面。

新增 | 返回商品列表

ID	所属商品	兑换码	购买人	兑换时间	是否已经兑换	
1	研磨设计模式	12234443232			false	删除
2	研磨设计模式	342342342342			false	删除
3	研磨设计模式	3423424234234			false	删除

图11-15 商品兑换码列表

3.4、JSP实现商品兑换码新增页面（**WEB-INF/jsp/admin/goodsCode/add.jsp**）

用于新增指定商品的兑换码。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="用户管理-新增"/>
</jsp:include>
<s:fielderror cssStyle="color:red"/>
<s:form action="/admin/goodsCode/add.action" method="POST" acceptcharset="UTF-8">
<s:token/>
<s:hidden name="goodsId" value="%{model.id}" />
<table border="1">
    <tr>
        <s:textfield label="所属商品" name="model.name" readonly="true"/>
    </tr>
    <tr>
        <s:textarea label="code码" name="codes" cols="20" rows="3"/>
    </tr>
    <tr>
       <td><input name="submit" type="submit" value="新增"/></td>
       <td><input name="cancel" type="button" onclick="javascript:window.location.href='$
    </tr>
</table>
</s:form>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Tomcat服务器，在商品兑换码列表中单击【新增】按钮将显示图11-16界面。

图11-16 兑换码新增页面

4、前台商品展示及购买模块：

前台商品展示提供商品展示及购买页面，购买时应考虑是否有足够兑换码等，此处错误消息使用硬编码，应该考虑使用国际化支持，请参考学习国际化。

4.1、**Action**实现

```
package cn.javass.point.web.front.action;
//省略import
public class GoodsAction extends BaseAction {
    private static final String BUY_RESULT = "buyResult";
    public String list() {
        getValueStack().set(PAGE, goodsService.listAllPublished(getPn()));
        return LIST;
    }
    public String buy() {
        String username = "test";
        GoodsCodeModel goodsCode = null;
        try {
            goodsCode = goodsCodeService.buy(username, goodsId);
        } catch (NotCodeException e) {
            this.addActionError("没有足够的兑换码了");
            return BUY_RESULT;
        } catch (Exception e) {
            e.printStackTrace();
            this.addActionError("未知错误");
            return BUY_RESULT;
        }
        this.addActionMessage("购买成功，您的兑换码为 :"+ goodsCode.getCode());
        getValueStack().set(MODEL, goodsCode);
        return BUY_RESULT;
    }
    //字段驱动数据填充
    private int goodsId;
    //省略字段驱动数据的getter和setter
    //依赖注入Service
    IGoodsService goodsService;
    IGoodsCodeService goodsCodeService;
    //省略依赖注入的getter和setter
}
```

4.2、Spring配置文件定义（resources/cn/javass/web/pointShop-front-servlet.xml）：

```
<bean name="/front/goodsAction" class="cn.javass.point.web.front.action.GoodsAction" scop
    <property name="goodsService" ref="goodsService"/>
    <property name="goodsCodeService" ref="goodsCodeService"/>
</bean>
```

4.3、JSP实现前台商品展示及购买页面（WEB-INF/jsp/ goods/list.jsp）

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="积分商城-商品列表"/>
</jsp:include>
<s:iterator value="page.items" status="status">
      <s:property value="#status.index + 1"/>.<s:property value="name"/>
      <a href="${ctx}/goods/buy.action?goodsId=<s:property value='id'/>">【购买】</a><br/:
     描述：<s:property value="description"/><br/>
      <s>需要积分<s:property value="originalPoint"/></s>  现需积分：<b><s:propert
</s:iterator>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Web服务器，在浏览器中输入http://localhost:8080/pointShop/goods/list.action将显示图11-17界面。

1.研磨设计模式 【购买】
描述：一本值得反复阅读的书
~~需要积分200~~ 现需积分：100
2.研磨设计模式 【购买】
描述：一本值得反复阅读的书
~~需要积分200~~ 现需积分：100
3.研磨设计模式 【购买】
描述：一本值得反复阅读的书
~~需要积分200~~ 现需积分：100

图11-17 前台商品展示即购买页面

在前台商品展示即购买页面中点击购买，如果库存中还有兑换码，将购买成功，否则购买失败。

4.3、商品购买结果页面（WEB-INF/jsp/admin/goods/buyResult.jsp）

购买成功将通过“<s:actionmessage/>”标签显示成功信息并将兑换码显示给用户，购买失败将通过“<s:actionerror/>”标签提示如积分不足或兑换码没有了等错误信息。

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="积分商城-购买结果"/>
</jsp:include>
<s:actionerror/>
<s:actionmessage/>
<jsp:include page="../../common/inc/footer.jsp"/>
```

在商品展示及购买列表购买成功或失败将显示图11-18或图11-19界面。

- 购买成功，您的兑换码为 :12234443232

图11-18 购买成功页面

- 没有足够的兑换码了

图11-19 购买失败页面

到此SSH集成已经结束，集成SSH是非常简单的，但开发流程及开发思想是关键。

我们整个开发过程是首先抽象和提取通用的模块和代码，这样可以复用减少开发时间，其次是基于通用层开发不可预测部分（即可变部分），因为每个项目的功能是不一样的。在开发过程中还集中将重复内容提取到一处这样方便以后修改。

【第十二章】零配置 之 12.1 概述 ——跟我学spring3

12.1 概述

12.1.1 什么是零配置

在SSH集成一章中大家注意到项目结构和包结构是不是很有规律，类库放到WEB-INF/lib文件夹下，jsp文件放到WEB-INF/jsp文件夹下，web.xml需要放到WEB-INF文件夹下等等，为什么要这么放呢？不这样放可以吗？

所谓零配置，并不是说一点配置都没有了，而是配置很少而已。通过约定来减少需要配置的数量，提高开发效率。

因此SSH集成时的项目结构和包结构完全是任意的，可以通过配置方式来指定位置，因此如web.xml完全可以不放在WEB-INF下边而通过如tomcat配置文件中新指定web.xml位置。

还有在SSH集成中还记得使用在Struts2配置文件中使用模式匹配通配符来定义action，只要我们的URL模式将类似于/{module}/{action}/{method}.action即可自动映射到相应的Action类的方法上，但如果你的URL不对肯定是映射不到的，这就是规约。

零配置并不是没有配置，而是通过约定来减少配置。那如何实现零配置呢？

12.1.2 零配置的实现方式

零配置实现主要有以下两种方式：

- 惯例优先原则：也称为约定大于配置或规约大于配置（convention over configuration），即通过约定代码结构或命名规范来减少配置数量，同样不会减少配置文件；即通过约定好默认规范来提高开发效率；如Struts2配置文件使用模式匹配通配符来定义action就是惯例优先原则。
- 基于注解的规约配置：通过在指定类上指定注解，通过注解约定其含义来减少配置数量，从而提高开发效率；如事务注解@Transaction是不是基于注解的规约，只有在指定的类或方法上使用该注解就表示其需要事务。

对惯例优先原则支持的有项目管理工具Maven，它约定了一套非常好的项目结构和一套合理的默认值来简化日常开发，作者比较喜欢使用Maven构建和管理项目；另外还有Strtus2的convention-plugin也提供了零配置支持等等。

大家还记得【7.5 集成Spring JDBC及最佳实践】时的80/20法则吗？零配置是不是同样很好的体现了这个法则，在日常开发中同样80%时间使用默认配置，而20%时间可能需要特定配置。

12.1.3 Spring3的零配置

Spring3中零配置的支持主要体现在Spring Web MVC框架的惯例优先原则和基于注解配置。

Spring Web MVC框架的惯例优先原则采用默认的命名规范来减少配置。

Spring基于注解的配置采用约定注解含义来减少配置，包括注解实现Bean配置、注解实现Bean定义和Java类替换配置文件三部分：

- 注解实现**Bean**依赖注入：通过注解方式替代基于XML配置中的依赖注入，如使用@Autowired注解来完成依赖注入。
- 注解实现**Bean**定义：通过注解方式进行Bean配置元数据定义，从而完全将Bean配置元数据从配置文件中移除。
- Java类替换配置文件：使用Java类来定义所有的Spring配置，完全消除XML配置文件。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/2543.html】

【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3

12.2 注解实现Bean依赖注入

12.2.1 概述

注解实现Bean配置主要用来进行如依赖注入、生命周期回调方法定义等，不能消除XML文件中的Bean元数据定义，且基于**XML**配置中的依赖注入的数据将覆盖基于注解配置中的依赖注入的数据。

Spring3的基于注解实现Bean依赖注入支持如下三种注解：

- **Spring**自带依赖注入注解： Spring自带的一套依赖注入注解；
- **JSR-250**注解：Java平台的公共注解，是Java EE 5规范之一，在JDK6中默认包含这些注解，从Spring2.5开始支持。
- **JSR-330**注解：Java 依赖注入标准，Java EE 6规范之一，可能在加入到未来JDK版本，从Spring3开始支持；
- **JPA**注解：用于注入持久化上下文和尸体管理器。

这三种类型的注解在Spring3中都支持，类似于注解事务支持，想要使用这些注解需要在Spring容器中开启注解驱动支持，即使用如下配置方式开启：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation=" http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>

</beans>
```

这样就能使用注解驱动依赖注入了，该配置文件位于“resources/chapter12/dependecyInjectWithAnnotation.xml”。

12.2.2 Spring自带依赖注入注解

一、**@Required**：依赖检查；

对应于基于XML配置中的依赖检查，但XML配置的依赖检查将检查所有setter方法，详见【3.3.4 依赖检查】；

基于@Required的依赖检查表示注解的setter方法必须，即必须通过在XML配置中配置setter注入，如果没有配置在容器启动时会抛出异常从而保证在运行时不会遇到空指针异常，@Required只能放置在setter方法上，且通过XML配置的setter注入，可以使用如下方式来指定：

```
@Requried
setter方法
```

1、准备测试Bean

```
package cn.javass.spring.chapter12;
public class TestBean {
    private String message;
    @Required
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean" class="cn.javass.spring.chapter12.TestBean">
<property name="message" ref="message"/>
</bean>
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

3、测试类和测试方法如下：

```
package cn.javass.spring.chapter12;
//省略import
public class DependencyInjectWithAnnotationTest {
    private static String configLocation = "classpath:chapter12/dependecyInjectWithAnnota
    private static ApplicationContext ctx = new ClassPathXmlApplicationContext(configLoca
    //1、Spring自带依赖注入注解
     @Test
    public void testRequiredForXmlSetterInject() {
        TestBean testBean = ctx.getBean("testBean", TestBean.class);
        Assert.assertEquals("hello", testBean.getMessage());
    }
}
```

在XML配置文件中必须指定setter注入，否则在Spring容器启动时将抛出如下异常：

```
org.springframework.beans.factory.BeanCreationException:
Error creating bean with name 'testBean' defined in class path resource [chapter12/depend
nested exception is org.springframework.beans.factory.BeanInitializationException: Proper
```

二、**@Autowired**：自动装配

自动装配，用于替代基于XML配置的自动装配，详见【3.3.3 自动装配】。

基于@Autowired的自动装配，默认是根据类型注入，可以用于构造器、字段、方法注入，使用方式如下：

```
@Autowired(required=true)
构造器、字段、方法
```

@Autowired默认是根据参数类型进行自动装配，且必须有一个Bean候选者注入，如果允许出现0个Bean候选者需要设置属性“required=false”，“required”属性含义和@Required一样，只是@Required只适用于基于XML配置的setter注入方式。

（**1**）、构造器注入：通过将@Autowired注解放在构造器上来完成构造器注入，默认构造器参数通过类型自动装配，如下所示：

1、准备测试Bean，在构造器上添加@AutoWired注解：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean11 {
    private String message;
    @Autowired //构造器注入
    private TestBean11(String message) {
        this.message = message;
    }
    //省略message的getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean11" class="cn.javass.spring.chapter12.TestBean11"/>
```

3、测试类如下：

```
@Test
public void testAutowiredForConstructor() {
    TestBean11 testBean11 = ctx.getBean("testBean11", TestBean11.class);
    Assert.assertEquals("hello", testBean11.getMessage());
}
```

在Spring配置文件中没有对“testBean11”进行构造器注入和setter注入配置，而是通过在构造器上添加@ Autowired来完成根据参数类型完成构造器注入。

（**2**）、字段注入：通过将@Autowired注解放在构造器上来完成字段注入。

1、准备测试Bean，在字段上添加@AutoWired注解：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean12 {
    @Autowired //字段注入
    private String message;
    //省略getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean12" class="cn.javass.spring.chapter12.TestBean12"/>
```

3、测试方法如下：

```
@Test
public void testAutowiredForField() {
    TestBean12 testBean12 = ctx.getBean("testBean12", TestBean12.class);
    Assert.assertEquals("hello", testBean12.getMessage());
}
```

字段注入在基于XML配置中无相应概念，字段注入不支持静态类型字段的注入。

（**3**）、方法参数注入：通过将@Autowired注解放在方法上来完成方法参数注入。

1、准备测试Bean，在方法上添加@AutoWired注解：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean13 {
    private String message;
    @Autowired //setter方法注入
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean14 {
    private String message;
    private List<String> list;
    @Autowired(required = true) //任意一个或多个参数方法注入
    private void initMessage(String message, ArrayList<String> list) {
        this.message = message;
        this.list = list;
    }
    //省略getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean13" class="cn.javass.spring.chapter12.TestBean13"/>
<bean id="testBean14" class="cn.javass.spring.chapter12.TestBean14"/>
<bean id="list" class="java.util.ArrayList">
    <constructor-arg index="0">
        <list>
            <ref bean="message"/>
            <ref bean="message"/>
        </list>
   </constructor-arg>
</bean>
```

3、测试方法如下：

```
@Test
public void testAutowiredForMethod() {
    TestBean13 testBean13 = ctx.getBean("testBean13", TestBean13.class);
    Assert.assertEquals("hello", testBean13.getMessage());

    TestBean14 testBean14 = ctx.getBean("testBean14", TestBean14.class);
    Assert.assertEquals("hello", testBean14.getMessage());
    Assert.assertEquals(ctx.getBean("list", List.class), testBean14.getList());
}
```

方法参数注入除了支持setter方法注入，还支持1个或多个参数的普通方法注入，在基于XML配置中不支持1个或多个参数的普通方法注入，方法注入不支持静态类型方法的注入。

注意“**initMessage(String message, ArrayList<String> list)**”方法签名中为什么使用ArrayList而不是List呢？具体参考【3.3.3 自动装配】一节中的集合类型注入区别。

三、**@Value**：注入**SpEL**表达式；

用于注入SpEL表达式，可以放置在字段方法或参数上，使用方式如下：

```
@Value(value = "SpEL表达式")
字段、方法、参数
```

1、可以在类字段上使用该注解：

```
@Value(value = "#{message}")
private String message;
```

2、可以放置在带@Autowired注解的方法的参数上：

```
@Autowired
public void initMessage(@Value(value = "#{message}#{message}") String message) {
    this.message = message;
}
```

3、还可以放置在带@Autowired注解的构造器的参数上：

```
@Autowired
private TestBean43(@Value(value = "#{message}#{message}") String message) {
    this.message = message;
}
```

具体测试详见DependencyInjectWithAnnotationTest 类的testValueInject测试方法。

四、**@Qualifier**：限定描述符，用于细粒度选择候选者；

@Autowired默认是根据类型进行注入的，因此如果有多个类型一样的Bean候选者，则需要限定其中一个候选者，否则将抛出异常，详见【3.3.3 自动装配】中的根据类型进行注入；

@Qualifier限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者，具体使用方式如下：

```
@Qualifier(value = "限定标识符")
字段、方法、参数
```

（**1**）、根据基于**XML**配置中的**<qualifier>**标签指定的名字进行注入，使用如下方式指定名称：

```
<qualifier  type="org.springframework.beans.factory.annotation.Qualifier"  value="限定标识
```

其中type属性可选，指定类型，默认就是Qualifier注解类，name就是给Bean候选者指定限定标识符，一个Bean定义中只允许指定类型不同的<qualifier>，如果有多个相同type后面指定的将覆盖前面的。

1、准备测试Bean：

```
package cn.javass.spring.chapter12;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class TestBean31 {
    private DataSource dataSource;
    @Autowired
    //根据<qualifier>标签指定Bean限定标识符
    public void initDataSource(@Qualifier("mysqlDataSource") DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean31" class="cn.javass.spring.chapter12.TestBean31"/>
```

我们使用@Qualifier("mysqlDataSource")来指定候选Bean的限定标识符，我们需要在配置文件中使用<qualifier>标签来指定候选Bean的限定标识符“mysqlDataSource”：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDa
     <qualifier value="mysqlDataSource"/>
</bean>
```

3、测试方法如下：

```
@Test
public void testQualifierInject1() {
    TestBean31 testBean31 = ctx.getBean("testBean31", TestBean31.class);
    try {
        //使用<qualifier>指定的标识符只能被@Qualifier使用
        ctx.getBean("mysqlDataSource");
        Assert.fail();
    } catch (Exception e) {
        //找不到该Bean
        Assert.assertTrue(e instanceof NoSuchBeanDefinitionException);
    }
     Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean31.getDataSource());
}
```

从测试可以看出使用<qualifier>标签指定的限定标识符只能被@Qualifier使用，不能作为Bean的标识符，如“ctx.getBean("mysqlDataSource")”是获取不到Bean的。

（**2**）、缺省的根据**Bean**名字注入：最基本方式，是在Bean上没有指定<qualifier>标签时一种容错机制，即缺省情况下使用Bean标识符注入，但如果你指定了<qualifier>标签将不会发生容错。

1、准备测试Bean：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean32 {
    private DataSource dataSource;
    @Autowired
    @Qualifier(value = "mysqlDataSource2") //指定Bean限定标识符
    //@Qualifier(value = "mysqlDataSourceBean")
    //是错误的注入，不会发生回退容错，因为你指定了<qualifier>
    public void initDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean32" class="cn.javass.spring.chapter12.TestBean32"/>
<bean id="oracleDataSource"  class="org.springframework.jdbc.datasource.DriverManagerData
```

3、测试方法如下：

```
@Test
public void testQualifierInject2() {
    TestBean32 testBean32 = ctx.getBean("testBean32", TestBean32.class);
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean32.getDataSource());
}
```

默认情况下（没指定<qualifier>标签）@Qualifier的value属性将匹配Bean 标识符。

（**3**）、扩展**@Qualifier**限定描述符注解：对@Qualifier的扩展来提供细粒度选择候选者；

具体使用方式就是自定义一个注解并使用@Qualifier注解其即可使用。

首先让我们考虑这样一个问题，如果我们有两个数据源，分别为Mysql和Oracle，因此注入两者相关资源时就牵扯到数据库相关，如在DAO层注入SessionFactory时，当然可以采用前边介绍的方式，但为了简单和直观我们希望采用自定义注解方式。

1、扩展@Qualifier限定描述符注解来分别表示Mysql和Oracle数据源

```
package cn.javass.spring.chapter12.qualifier;
import org.springframework.beans.factory.annotation.Qualifier;
/** 表示注入Mysql相关 */
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Mysql {
}
```

```
package cn.javass.spring.chapter12.qualifier;
import org.springframework.beans.factory.annotation.Qualifier;
/** 表示注入Oracle相关 */
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Oracle {
}
```

2、准备测试Bean：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean33 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Autowired
    public void initDataSource(@Mysql DataSource mysqlDataSource, @Oracle DataSource orac
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
    }
    public DataSource getMysqlDataSource() {
        return mysqlDataSource;
    }
    public DataSource getOracleDataSource() {
        return oracleDataSource;
    }
}
```

3、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean33" class="cn.javass.spring.chapter12.TestBean33"/>
```

4、在Spring修改定义的两个数据源：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDa
     <qualifier value="mysqlDataSource"/>
     <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
</bean>
<bean id="oracleDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataS
      <qualifier type="cn.javass.spring.chapter12.qualifier.Oracle"/>
</bean>
```

5、测试方法如下：

```
@Test
public void testQualifierInject3() {
    TestBean33 testBean33 = ctx.getBean("testBean33", TestBean33.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean33.getMysqlDataSoruce
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean33.getOracleDataSoruce()
}
```

测试也通过了，说明我们扩展的@Qualifier限定描述符注解也能很好工作。

前边演示了不带属性的注解，接下来演示一下带参数的注解：

1、首先定义数据库类型：

```
package cn.javass.spring.chapter12.qualifier;
public enum DataBase {
    ORACLE, MYSQL;
}
```

2、其次扩展@Qualifier限定描述符注解

```
package cn.javass.spring.chapter12.qualifier;
//省略import
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DataSourceType {
    String ip();      //指定ip,用于多数据源情况
    DataBase database();//指定数据库类型
}
```

3、准备测试Bean：

```
package cn.javass.spring.chapter12;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import cn.javass.spring.chapter12.qualifier.DataBase;
import cn.javass.spring.chapter12.qualifier.DataSourceType;
public class TestBean34 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Autowired
    public void initDataSource(
            @DataSourceType(ip="localhost", database=DataBase.MYSQL)
            DataSource mysqlDataSource,
            @DataSourceType(ip="localhost", database=DataBase.ORACLE)
            DataSource oracleDataSource) {
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
    }
      //省略getter方法
}
```

4、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean34" class="cn.javass.spring.chapter12.TestBean34"/>
```

5、在Spring修改定义的两个数据源：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDa
    <qualifier value="mysqlDataSource"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
        <attribute key="ip" value="localhost"/>
        <attribute key="database" value="MYSQL"/>
    </qualifier>
</bean>
<bean id="oracleDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataS
    <qualifier type="cn.javass.spring.chapter12.qualifier.Oracle"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
        <attribute key="ip" value="localhost"/>
        <attribute key="database" value="ORACLE"/>
    </qualifier>
</bean>
```

6、测试方法如下：

```
@Test
public void testQualifierInject3() {
    TestBean34 testBean34 = ctx.getBean("testBean34", TestBean34.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean34.getMysqlDataSource
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean34.getOracleDataSoruce()
}
```

测试也通过了，说明我们扩展的@Qualifier限定描述符注解也能很好工作。

四、自定义注解限定描述符：完全不使用@Qualifier，而是自己定义一个独立的限定注解；

1、首先使用如下方式定义一个自定义注解限定描述符：

```
package cn.javass.spring.chapter12.qualifier;
//省略import
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface CustomQualifier {
    String value();
}
```

2、准备测试Bean：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean35 {
    private DataSource dataSoruce;
    @Autowired
    public TestBean35(@CustomQualifier("oracleDataSource") DataSource dataSource) {
        this.dataSoruce = dataSource;
    }
    public DataSource getDataSoruce() {
        return dataSoruce;
    }
}
```

3、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean35" class="cn.javass.spring.chapter12.TestBean35"/>
```

4、然后在Spring配置文件中注册CustomQualifier自定义注解限定描述符，只有注册了Spring才能识别：

```
<bean id="customAutowireConfigurer" class="org.springframework.beans.factory.annotation.C
    <property name="customQualifierTypes">
        <set>
            <value>cn.javass.spring.chapter12.qualifier.CustomQualifier</value>
        </set>
   </property>
</bean>
```

5、测试方法如下：

```
@Test
public void testQualifierInject5() {
    TestBean35 testBean35 = ctx.getBean("testBean35", TestBean35.class);
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean35.getDataSource());
}
```

从测试中可看出，自定义的和Spring自带的没什么区别，因此如果没有足够的理由请使用Spring自带的Qualifier注解。

到此限定描述符介绍完毕，在此一定要注意以下几点：

- 限定标识符和Bean的描述符是不一样的；
- 多个Bean定义中可以使用相同的限定标识符；
- 对于集合、数组、字典类型的限定描述符注入，将注入多个具有相同限定标识符的Bean。

12.2.3 JSR-250注解

一、**@Resource**：自动装配，默认根据类型装配，如果指定name属性将根据名字装配，可以使用如下方式来指定：

```
@Resource(name = "标识符")
字段或setter方法
```

1、准备测试Bean：

```
package cn.javass.spring.chapter12;
import javax.annotation.Resource;
public class TestBean41 {
    @Resource(name = "message")
    private String message;
    //省略getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean41" class="cn.javass.spring.chapter12.TestBean41"/>
```

3、测试方法如下：

```
@Test
public void testResourceInject1() {
    TestBean41 testBean41 = ctx.getBean("testBean41", TestBean41.class);
    Assert.assertEquals("hello", testBean41.getMessage());
}
```

使用非常简单，和@Autowired不同的是可以指定name来根据名字注入。

使用@Resource需要注意以下几点：

- @Resource注解应该只用于setter方法注入，不能提供如@Autowired多参数方法注入；
- @Resource在没有指定name属性的情况下首先将根据setter方法对于的字段名查找资源，如果找不到再根据类型查找；
- @Resource首先将从JNDI环境中查找资源，如果没找到默认再到Spring容器中查找，因此如果JNDI环境中有和Spring容器同名的资源时需要注意。

二、**@PostConstruct**和**PreDestroy**：通过注解指定初始化和销毁方法定义；

1、在测试类TestBean41中添加如下代码：

```
@PostConstruct
public void init() {
    System.out.println("==========init");
}
@PreDestroy
public void destroy() {
    System.out.println("==========destroy");
}
```

2、修改测试方法如下：

```
@Test
public void resourceInjectTest1() {
    ((ClassPathXmlApplicationContext) ctx).registerShutdownHook();
    TestBean41 testBean41 = ctx.getBean("testBean41", TestBean41.class);
    Assert.assertEquals("hello", testBean41.getMessage());
}
```

类似于通过<bean>标签的init-method和destroy-method属性指定的初始化和销毁方法，但具有更高优先级，即注解方式的初始化和销毁方法将先执行。

12.2.4 JSR-330注解

在测试之前需要准备JSR-330注解所需要的jar包，到spring-framework-3.0.5.RELEASE-dependencies.zip中拷贝如下jar包到类路径：

```
com.springsource.javax.inject-1.0.0.jar
```

一、**@Inject**：等价于默认的@Autowired，只是没有required属性；

二、**@Named**：指定Bean名字，对应于Spring自带@Qualifier中的缺省的根据Bean名字注入情况；

三、**@Qualifier**：只对应于Spring自带@Qualifier中的扩展@Qualifier限定描述符注解，即只能扩展使用，没有value属性。

1、首先扩展@Qualifier限定描述符注解来表示Mysql数据源

```
package cn.javass.spring.chapter12.qualifier;
//省略部分import
import javax.inject.Qualifier;
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface JSR330Mysql {
}
```

2、准备测试Bean：

```
package cn.javass.spring.chapter12;
import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;
import cn.javass.spring.chapter12.qualifier.JSR330Mysql;
public class TestBean51 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Inject
    public void initDataSoruce(
            @JSR330Mysql  DataSource mysqlDataSource,
            @Named("oracleDataSource") DataSource oracleDataSource) {
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;

    }
     //省略getter
}
```

3、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<bean id="testBean51" class="cn.javass.spring.chapter12.TestBean51"/>
```

4、在Spring修改定义的mysqlDataSourceBean数据源：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDa
         <qualifier value="mysqlDataSource"/>
         <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
         <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
             <attribute key="ip" value="localhost"/>
             <attribute key="database" value="MYSQL"/>
         </qualifier>
         <qualifier type="cn.javass.spring.chapter12.qualifier.JSR330Mysql"/>
</bean>
```

5、测试方法如下：

```
@Test
public void testInject() {
    TestBean51 testBean51 = ctx.getBean("testBean51", TestBean51.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean51.getMysqlDataSource
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean51.getOracleDataSource()
}
```

测试也通过了，说明JSR-330注解也能很好工作。

从测试中可以看出JSR-330注解和Spring自带注解依赖注入时主要有以下特点：

- Spring自带的@Autowired的缺省情况等价于JSR-330的@Inject注解；
- Spring自带的@Qualifier的缺省的根据Bean名字注入情况等价于JSR-330的@Named注解；
- Spring自带的@Qualifier的扩展@Qualifier限定描述符注解情况等价于JSR-330的@Qualifier注解。

12.2.5 JPA注解

用于注入EntityManagerFactory和EntityManager。

1、准备测试Bean：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean61 {
    @PersistenceContext(unitName = "entityManagerFactory")
    private EntityManager entityManager;

    @PersistenceUnit(unitName = "entityManagerFactory")
    private EntityManagerFactory entityManagerFactory;

    public EntityManager getEntityManager() {
        return entityManager;
    }
    public EntityManagerFactory getEntityManagerFactory() {
        return entityManagerFactory;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

```
<import resource="classpath:chapter7/applicationContext-resources.xml"/>
<import resource="classpath:chapter8/applicationContext-jpa.xml"/>
<bean id="testBean61" class="cn.javass.spring.chapter12.TestBean61"/>
```

此处需要引用第七章和八章的配置文件，细节内容请参考七八两章。

3、测试方法如下：

```
@Test
public void testJpaInject() {
    TestBean61 testBean61 = ctx.getBean("testBean61", TestBean61.class);
    Assert.assertNotNull(testBean61.getEntityManager());
    Assert.assertNotNull(testBean61.getEntityManagerFactory());
}
```

测试也通过了，说明JPA注解也能很好工作。

JPA注解类似于@Resource注解同样是先根据unitName属性去JNDI环境中查找，如果没找到在到Spring容器中查找。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/2545.html】

【第十二章】零配置 之 12.3 注解实现Bean定义 ——跟我学spring3

12.3 注解实现Bean定义

12.3.1 概述

前边介绍的Bean定义全是基于XML方式定义配置元数据，且在【12.2注解实现Bean依赖注入】一节中介绍了通过注解来减少配置数量，但并没有完全消除在XML配置文件中的Bean定义，因此有没有方式完全消除XML配置Bean定义呢？

Spring提供通过扫描类路径中的特殊注解类来自动注册Bean定义。同注解驱动事务一样需要开启自动扫描并注册Bean定义支持，使用方式如下（resources/chapter12/componentDefinitionWithAnnotation.xml）：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <aop:aspectj-autoproxy />

    <context:component-scan base-package="cn.javass.spring.chapter12"/>

</beans>
```

使用<context:component-scan>标签来表示需要要自动注册Bean定义，而通过base-package属性指定扫描的类路径位置。

<context:component-scan>标签将自动开启"注解实现**Bean**依赖注入"支持。

此处我们还通过<aop:aspectj-autoproxy/>用于开启Spring对@AspectJ风格切面的支持。

Spring基于注解实现Bean定义支持如下三种注解：

- **Spring**自带的**@Component**注解及扩展**@Repository**、**@Service**、**@Controller**，如图12-1所示；
- **JSR-250 1.1**版本中中定义的**@ManagedBean**注解，是Java EE 6标准规范之一，不包括在JDK中，需要在应用服务器环境使用（如Jboss），如图12-2所示；
- **JSR-330**的**@Named**注解，如图12-3所示。

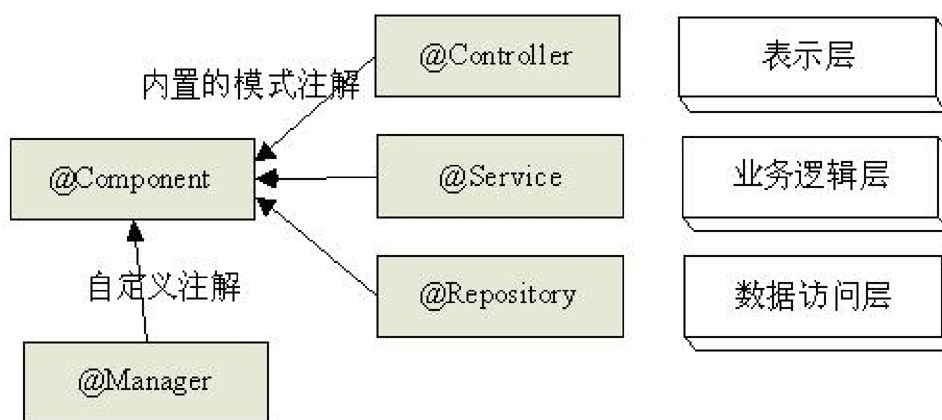


图12-1 Spring自带的@Component注解及扩展

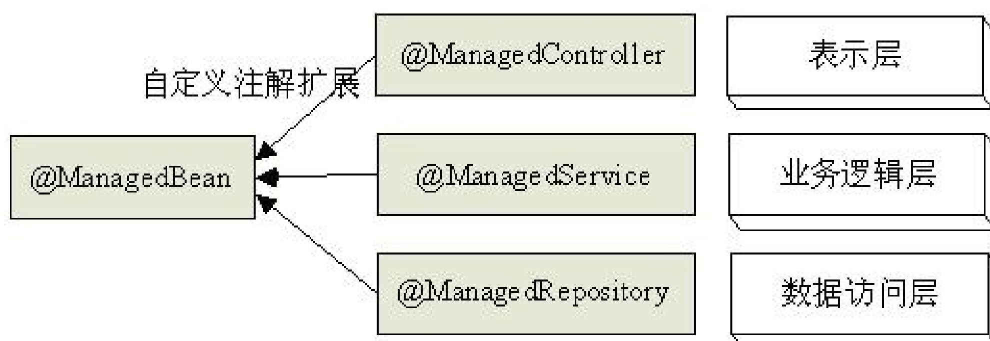


图12-2 JSR-250中定义的@ManagedBean注解及自定义扩展

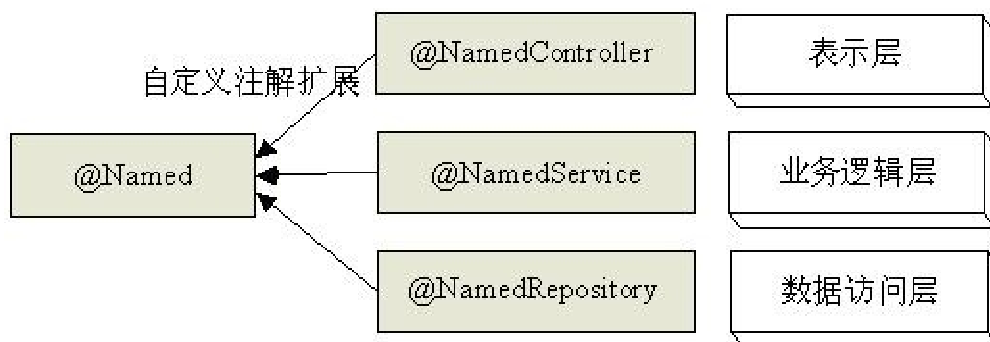


图12-3 JSR-330的@Named注解及自定义扩展

图12-2和图12-3中的自定义扩展部分是为了配合Spring自带的模式注解扩展自定义的，并不包含在Java EE 6规范中，在Java EE 6中相应的服务层、DAO层功能由EJB来完成。

在Java EE中有些注解运行放置在多个地方，如@Named允许放置在类型、字段、方法参数上等，因此一般情况下放置在类型上表示定义，放置在参数、方法等上边一般代表使用（如依赖注入等等）。

12.3.2 Spring自带的@Component注解及扩展

一、**@Component**：定义**Spring**管理**Bean**，使用方式如下：

```
@Component("标识符")
POJO类
```

在类上使用@Component注解，表示该类定义为Spring管理Bean，使用默认value（可选）属性表示Bean标识符。

1、定义测试Bean类:

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;
@Component("component")
public class TestCompoment {
    @Autowired
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试类和测试方法：

```
package cn.javass.spring.chapter12;
//省略import
public class ComponentDefinitionWithAnnotationTest {
    private static String configLocation = "classpath:chapter12/componentDefinitionWithAn
    private static ApplicationContext ctx = new ClassPathXmlApplicationContext(configLoca
    @Test
    public void testComponent() {
        TestCompoment component = ctx.getBean("component", TestCompoment.class);
        Assert.assertNotNull(component.getCtx());
    }
}
```

测试成功说明被@Component注解的POJO类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配。

@AspectJ风格的切面可以通过**@Compenent**注解标识其为**Spring**管理**Bean**，而**@Aspect**注解不能被**Spring**自动识别并注册为**Bean**，必须通过**@Component**注解来完成，示例如下：

```
package cn.javass.spring.chapter12.aop;
//省略import
@Component
@Aspect
public class TestAspect {
    @Pointcut(value="execution(* *(..))")
    private void pointcut() {}
    @Before(value="pointcut()")
    public void before() {
        System.out.println("=======before");
    }
}
```

通过**@Component**将切面定义为**Spring**管理**Bean**。

二、**@Repository**：**@Component**扩展，被**@Repository**注解的**POJO**类表示**DAO**层实现，从而见到该注解就想到**DAO**层实现，使用方式和**@Component**相同；

1、定义测试Bean类:

```
package cn.javass.spring.chapter12.dao.hibernate;
import org.springframework.stereotype.Repository;
@Repository("testHibernateDao")
public class TestHibernateDaoImpl {

}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

```
@Test
public void testDao() {
TestHibernateDaoImpl dao =
ctx.getBean("testHibernateDao", TestHibernateDaoImpl.class);
Assert.assertNotNull(dao);
}
```

测试成功说明被@Repository注解的POJO类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配，并且被@Repository注解的类表示DAO层实现。

三、**@Service**：**@Component**扩展，被**@Service**注解的**POJO**类表示**Service**层实现，从而见到该注解就想到**Service**层实现，使用方式和**@Component**相同；

1、定义测试Bean类:

```
package cn.javass.spring.chapter12.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import cn.javass.spring.chapter12.dao.hibernate.TestHibernateDaoImpl;
@Service("testService")
public class TestServiceImpl {
    @Autowired
    @Qualifier("testHibernateDao")
    private TestHibernateDaoImpl dao;
    public TestHibernateDaoImpl getDao() {
        return dao;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

```
@Test
public void testService() {
    TestServiceImpl service = ctx.getBean("testService", TestServiceImpl.class);
    Assert.assertNotNull(service.getDao());
}
```

测试成功说明被@Service注解的POJO类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配，并且被@Service注解的类表示Service层实现。

四、**@Controller**：**@Component**扩展，被**@Controller**注解的类表示**Web**层实现，从而见到该注解就想到**Web**层实现，使用方式和**@Component**相同；

1、定义测试Bean类:

```
package cn.javass.spring.chapter12.action;
//省略import
@Controller
public class TestAction {
    @Autowired
    private TestServiceImpl testService;

    public void list() {
        //调用业务逻辑层方法
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

```
@Test
public void testWeb() {
    TestAction action = ctx.getBean("testAction", TestAction.class);
    Assert.assertNotNull(action);
}
```

测试成功说明被@Controller注解的类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配，并且被@Controller注解的类表示Web层实现。

大家是否注意到@Controller中并没有定义Bean的标识符，那么默认Bean的名字将是以小写开头的类名（不包括包名），即如“TestAction”类的Bean标识符为“testAction”。

六、自定义扩展：**Spring**内置了三种通用的扩展注解**@Repository**、**@Service**、**@Controller** ，大多数情况下没必要定义自己的扩展，在此我们演示下如何扩展**@Component**注解来满足某些特殊规约的需要；

在此我们可能需要一个缓存层用于定义缓存Bean，因此我们需要自定义一个@Cache的注解来表示缓存类。

1、扩展@Component：

```
package cn.javass.spring.chapter12.stereotype;
//省略import
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Cache{
       String value() default "";
}
```

扩展十分简单，只需要在扩展的注解上注解@Component即可，**@Repository**、@Service、@Controller也是通过该方式实现的，没什么特别之处

2、定义测试Bean类:

```
package cn.javass.spring.chapter12.cache;
@Cache("cache")
public class TestCache {

}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

```
@Test
public void testCache() {
    TestCache cache = ctx.getBean("cache", TestCache.class);
    Assert.assertNotNull(cache);
}
```

测试成功说明自定义的@Cache注解也能很好的工作，而且实现了我们的目的，使用@Cache来表示被注解的类是Cache层Bean。

12.3.3 JSR-250中定义的@ManagedBean注解

@javax.annotation.ManagedBean需要在实现Java EE 6规范的应用服务器上使用，虽然Spring3实现了，但@javax.annotation.ManagedBean只有在Java EE 6环境中才有定义，因此测试前需要我们定义ManagedBean类。

1、定义javax.annotation.ManagedBean注解类：

```
package javax.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ManagedBean {
     String value() default "";
}
```

其和@Component完全相同，唯一不同的就是名字和创建者（一个是Spring，一个是Java EE规范）。

2、定义测试Bean类:

```
package cn.javass.spring.chapter12;
import javax.annotation.Resource;
import org.springframework.context.ApplicationContext;
@javax.annotation.ManagedBean("managedBean")
public class TestManagedBean {
    @Resource
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

```
@Test
public void testManagedBean() {
    TestManagedBean testManagedBean = ctx.getBean("managedBean", TestManagedBean.class);
    Assert.assertNotNull(testManagedBean.getCtx());
}
```

测试成功说明被@ManagedBean注解类也能正常工作。

自定义扩展就不介绍了，大家可以参考@Component来完成如图12-2所示的自定义扩展部分。

12.3.4 JSR-330的@Named注解

@Named不仅可以用于依赖注入来指定注入的Bean的标识符，还可以用于定义Bean。即注解在类型上表示定义Bean，注解在非类型上（如字段）表示指定依赖注入的Bean标识符。

1、定义测试Bean类:

```
package cn.javass.spring.chapter12;
//省略import
@Named("namedBean")
public class TestNamedBean {
    @Inject
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

```
@Test
public void testNamedBean() {
TestNamedBean testNamedBean =
    ctx.getBean("namedBean", TestNamedBean.class);
    Assert.assertNotNull(testNamedBean.getCtx());
}
```

测试成功说明被@Named注解类也能正常工作。

自定义扩展就不介绍了，大家可以参考@Component来完成如图12-3所示的自定义扩展部分。

12.3.5 细粒度控制Bean定义扫描

在XML配置中完全消除了Bean定义，而是只有一个<context:component-scan>标签来支持注解Bean定义扫描。

前边的示例完全采用默认扫描设置，如果我们有几个组件不想被扫描并自动注册、我们想更改默认的Bean标识符生成策略该如何做呢？接下来让我们看一下如何细粒度的控制Bean定义扫描，具体定义如下：

```
<context:component-scan
        base-package=""
        resource-pattern="**/*.class"
        name-generator="org.springframework.context.annotation.AnnotationBeanNameGenerato
        use-default-filters="true"
        annotation-config="true">
                <context:include-filter type="aspectj" expression=""/>
                <context:exclude-filter type="regex" expression=""/>
</context:component-scan>
```

- **base-package**：表示扫描注解类的开始位置，即将在指定的包中扫描，其他包中的注解类将不被扫描，默认将扫描所有类路径；
- **resource-pattern**：表示扫描注解类的后缀匹配模式，即“base-package+resource-pattern”将组成匹配模式用于匹配类路径中的组件，默认后缀为“*/.class”，即指定包下的所有以.class结尾的类文件；
- **name-generator**：默认情况下的**Bean**标识符生成策略，默认是AnnotationBeanNameGenerator，其将生成以小写开头的类名（不包括包名）；可以自定义自己的标识符生成策略；
- **use-default-filters**：默认为true表示过滤@Component、@ManagedBean、@Named注解的类，如果改为false默认将不过滤这些默认的注解来定义Bean，即这些注解类不能被过滤到，即不能通过这些注解进行Bean定义；
- **annotation-config**：表示是否自动支持注解实现Bean依赖注入，默认支持，如果设置为false，将关闭支持注解的依赖注入，需要通过<context:annotation-config/>开启。

默认情况下将自动过滤@Component、@ManagedBean、@Named注解的类并将其注册为Spring管理Bean，可以通过在<context:component-scan>标签中指定自定义过滤器将过滤到匹配条件的类注册为Spring管理Bean，具体定义方式如下：

```
<context:include-filter type="aspectj" expression=""/>
<context:exclude-filter type="regex" expression=""/>
```

- **<context:include-filter>**：表示过滤到的类将被注册为Spring管理Bean；
- **<context:exclude-filter>**：表示过滤到的类将不被注册为Spring管理Bean，它比<context:include-filter>具有更高优先级；
- **type**：表示过滤器类型，目前支持注解类型、类类型、正则表达式、aspectj表达式过滤器，当然也可以自定义自己的过滤器，实现org.springframework.core.type.filter.TypeFilter即可；
- **expression**：表示过滤器表达式。

一般情况下没必要进行自定义过滤，如果需要请参考如下示例：

1、cn.javass.spring.chapter12.TestBean14自动注册为Spring管理Bean：

```
<context:include-filter type="assignable" expression="cn.javass.spring.chapter12.TestBean
```

2、把所有注解为org.aspectj.lang.annotation.Aspect自动注册为Spring管理Bean：

```
<context:include-filter type="annotation"
expression="org.aspectj.lang.annotation.Aspect"/>
```

3、将把匹配到正则表达式“cn.javass.spring.chapter12.TestBean2*”排除，不注册为Spring管理Bean：

```
<context:exclude-filter type="regex" expression="cn\.javass\.spring\.chapter12\.TestBean2
```

4、将把匹配到aspectj表达式“cn.javass.spring.chapter12.TestBean3*”排除，不注册为Spring管理Bean：

```
<context:exclude-filter type="aspectj" expression="cn.javass.spring.chapter12.TestBean3*"
```

具体使用就要看项目需要了，如果以上都不满足需要请考虑使用自定义过滤器。

12.3.6 提供更多的配置元数据

1、@Lazy：定义Bean将延迟初始化，使用方式如下：

```
@Component("component")
@Lazy(true)
public class TestCompoment {
……
}
```

使用@Lazy注解指定Bean需要延迟初始化。

2、**@DependsOn**：定义Bean初始化及销毁时的顺序，使用方式如下：

```
@Component("component")
@DependsOn({"managedBean"})
public class TestCompoment {
……
}
```

3、@Scope：定义Bean作用域，默认单例，使用方式如下：

```
@Component("component")
@Scope("singleton")
public class TestCompoment {
……
}
```

4、@Qualifier：指定限定描述符，对应于基于XML配置中的<qualifier>标签，使用方式如下：

```
@Component("component")
@Qualifier("component")
public class TestCompoment {
……
}
```

可以使用复杂的扩展，如@Mysql等等。

5、@Primary：自动装配时当出现多个Bean候选者时，被注解为@Primary的Bean将作为首选者，否则将抛出异常，使用方式如下：

```
@Component("component")
@Primary
public class TestCompoment {
……
}
```

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/2547.html】

【第十二章】零配置 之 12.4 基于Java类定义Bean配置元数据 ——跟我学spring3

12.4 基于Java类定义Bean配置元数据

12.4.1 概述

基于Java类定义Bean配置元数据，其实就是通过Java类定义Spring配置元数据，且直接消除XML配置文件。

基于Java类定义Bean配置元数据中的@Configuration注解的类等价于XML配置文件，@Bean注解的方法等价于XML配置文件中的Bean定义。

基于Java类定义Bean配置元数据需要通过AnnotationConfigApplicationContext加载配置类及初始化容器，类似于XML配置文件需要使用ClassPathXmlApplicationContext加载配置文件及初始化容器。

基于Java类定义Bean配置元数据需要CGLIB的支持，因此要保证类路径中包括CGLIB的jar包。

12.4.2 Hello World

首先让我们看一下基于Java类如何定义Bean配置元数据，具体步骤如下：

1、 通过@Configuration注解需要作为配置的类，表示该类将定义Bean配置元数据；

2、 通过@Bean注解相应的方法，该方法名默认就是Bean名，该方法返回值就是Bean对象；

3、 通过AnnotationConfigApplicationContext或子类加载基于Java类的配置。

接下来让我们先来学习一下如何通过Java类定义Bean配置元数据吧：

1、定义配置元数据的Java类如下所示：

```
package cn.javass.spring.chapter12.configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class ApplicationContextConfig {
    @Bean
    public String message() {
        return "hello";
    }
}
```

2、定义测试类，测试一下Java配置类是否工作：

```
package cn.javass.spring.chapter12.configuration;
//省略import
public class ConfigurationTest {
    @Test
    public void testHelloworld () {
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(A
        Assert.assertEquals("hello", ctx.getBean("message"));
    }
}
```

测试没有报错说明测试通过了，那AnnotationConfigApplicationContext是如何工作的呢，接下来让我们分析一下：

- 使用@Configuration注解配置类，该配置类定义了Bean配置元数据；
- 使用@Bean注解配置类中的方法，该方法名就是Bean的名字，该方法返回值就是Bean对象。
- 使用new AnnotationConfigApplicationContext(ApplicationContextConfig.class)创建应用上下文，构造器参数为使用@Configuration注解的配置类，读取配置类进行实例化相应的Bean。

知道如何使用了，接下来就详细介绍每个部分吧。

12.4.3 @Configuration

通过@Configuration注解的类将被作为配置类使用，表示在该类中将定义Bean配置元数据，且使用@Configuration注解的类本身也是一个Bean，使用方式如下所示：

```
import org.springframework.context.annotation.Configuration;
@Configuration("ctxConfig")
public class ApplicationContextConfig {
    //定义Bean配置元数据
}
```

因为使用@Configuration注解的类本身也是一个Bean，因为@Configuration被@Component注解了，因此@Configuration注解可以指定value属性值，如“ctxConfig”就是该Bean的名字，如使用“ctx.getBean("ctxConfig")”将返回该Bean。

使用@Configuration注解的类不能是final的，且应该有一个默认无参构造器。

12.4.4 @Bean

通过@Bean注解配置类中的相应方法，则该方法名默认就是Bean名，该方法返回值就是Bean对象，并定义了Spring IoC容器如何实例化、自动装配、初始化Bean逻辑，具体使用方法如下：

```
@Bean(name={},
      autowire=Autowire.NO,
      initMethod="",
      destroyMethod="")
```

- **name**：指定Bean的名字，可有多个，第一个作为Id，其他作为别名；
- **autowire**：自动装配，默认no表示不自动装配该Bean，另外还有Autowire.BY_NAME表示根据名字自动装配，Autowire.BY_TYPE表示根据类型自动装配；
- **initMethod**和**destroyMethod**：指定Bean的初始化和销毁方法。

示例如下所示（ApplicationContextConfig.java）

```
@Bean
public String message() {
    return new String("hello");
}
```

如上使用方式等价于如下基于XML配置方式

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

使用@Bean注解的方法不能是private、final或static的。

12.4.5 提供更多的配置元数据

详见【12.3.6 提供更多的配置元数据】中介绍的各种注解，这些注解同样适用于@Bean注解的方法。

12.4.6 依赖注入

基于Java类配置方式的Bean依赖注入有如下两种方式：

- 直接依赖注入，类似于基于XML配置方式中的显示依赖注入；
- 使用注解实现Bean依赖注入：如@Autowired等等。

在本示例中我们将使用【第三章 DI】中的测试Bean。

1、直接依赖注入：包括构造器注入和setter注入。

- 构造器注入：通过在@Bean注解的实例化方法中使用有参构造器实例化相应的Bean即可，如下所示(ApplicationContextConfig.java)：

```
@Bean
public HelloApi helloImpl3() {
    //通过构造器注入,分别是引用注入（message()）和常量注入（1）
    return new HelloImpl3(message(), 1); //测试Bean详见【3.1.2  构造器注入】
}
```

- **setter**注入：通过在@Bean注解的实例化方法中使用无参构造器实例化后，通过相应的setter方法注入即可，如下所示(ApplicationContextConfig.java)：

```
@Bean
public HelloApi helloImpl4() {
    HelloImpl4 helloImpl4 = new HelloImpl4();//测试Bean详见【3.1.3  setter注入】
    //通过setter注入注入引用
    helloImpl4.setMessage(message());
    //通过setter注入注入常量
    helloImpl4.setIndex(1);
    return helloImpl4;
}
```

2、使用注解实现**Bean**依赖注入：详见【12.2 注解实Bean依赖注入】。

具体测试方法如下(ConfigurationTest.java)：

```
@Test
public void testDependencyInject() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Appli
    ctx.getBean("helloImpl3", HelloApi.class).sayHello();
    ctx.getBean("helloImpl4", HelloApi.class).sayHello();
}
```

12.4.7 方法注入

在基于XML配置方式中，Spring支持查找方法注入和替换方法注入，但在基于Java配置方式中只支持查找方法注入，一般用于在一个单例Bean中注入一个原型Bean的情况，具体详见【3.3.5 方法注入】，如下所示（ApplicationContextConfig.java）：

```
@Bean
@Scope("singleton")
public HelloApi helloApi2() {
    HelloImpl5 helloImpl5 = new HelloImpl5() {
        @Override
        public Printer createPrototypePrinter() {
            //方法注入，注入原型Bean
            return prototypePrinter();
        }
        @Override
        public Printer createSingletonPrinter() {
            //方法注入，注入单例Bean
            return singletonPrinter();
        }
    };
    //依赖注入,注入单例Bean
    helloImpl5.setPrinter(singletonPrinter());
    return helloImpl5;
}
```

```
@Bean
@Scope(value="prototype")
public Printer prototypePrinter() {
    return new Printer();
 }
@Bean
@Scope(value="singleton")
public Printer singletonPrinter() {
    return new Printer();
}
```

具体测试方法如下(ConfigurationTest.java)：

```
@Test
public void testLookupMethodInject() {
    AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
    System.out.println("=======prototype sayHello======");
    HelloApi helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
    helloApi2.sayHello();
    helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
    helloApi2.sayHello();
}
```

如上测试等价于【3.3.5 方法注入】中的查找方法注入。

12.4.8 @Import

类似于基于XML配置中的<import/>，基于Java的配置方式提供了@Import来组合模块化的配置类，使用方式如下所示：

```
package cn.javass.spring.chapter12.configuration;
//省略import
@Configuration("ctxConfig2")
@Import({ApplicationContextConfig.class})
public class ApplicationContextConfig2 {
    @Bean(name = {"message2"})
    public String message() {
        return "hello";
    }
}
```

具体测试方法如下(ConfigurationTest.java)：

```
@Test
public void  importTest() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Appli
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

使用非常简单，在此就不多介绍了。

12.4.9 结合基于Java和基于XML方式的配置

基于Java方式的配置方式不是为了完全替代基于XML方式的配置，两者可以结合使用，因此可以有两种结合使用方式：

- 在基于Java方式的配置类中引入基于XML方式的配置文件；
- 在基于XML方式的配置文件中中引入基于Java方式的配置。

一、在基于**Java**方式的配置类中引入基于**XML**方式的配置文件：在@Configuration注解的配置类上通过@ImportResource注解引入基于XML方式的配置文件，示例如下所示：

1、定义基于XML方式的配置文件(chapter12/configuration/importResource.xml)：

```
<bean id="message3" class="java.lang.String">
    <constructor-arg index="0" value="test"></constructor-arg>
</bean>
```

2、修改基于Java方式的配置类ApplicationContextConfig，添加如下注解：

```
@Configuration("ctxConfig") //1、使用@Configuration注解配置类
@ImportResource("classpath:chapter12/configuration/importResource.xml")
public class ApplicationContextConfig {
……
}
```

使用@ImportResource引入基于XML方式的配置文件，如果有多个请使用@ImportResource({"config1.xml", "config2.xml"})方式指定多个配置文件。

二、在基于**XML**方式的配置文件中中引入基于**Java**方式的配置：直接在XML配置文件中声明使用@Configuration注解的配置类即可，示例如下所示：

1、定义基于Java方式的使用@Configuration注解的配置类在此我们使用ApplicationContextConfig.java。

2、定义基于XML方式的配置文件（chapter12/configuration/xml-config.xml）：

```
<context:annotation-config/>
<bean id="ctxConfig" class="cn.javass.spring.chapter12.configuration.ApplicationContextCo
```

- <context:annotation-config/>：用于开启对注解驱动支持，详见【12.2 注解实现Bean依赖注入】；
- <bean id="ctxConfig" class="……"/>：直接将使用@Configuration注解的配置类在配置文件中进行Bean定义即可。

3、测试代码如下所示(ConfigurationTest.java)：：

```
public void testXmlConfig() {
    String configLocations[] = {"chapter12/configuration/xml-config.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

测试成功，说明通过在基于XML方式的配置文件中能获取到基于Java方式的配置文件中定义的Bean，如“message”Bean。

12.4.10 基于Java方式的容器实例化

基于Java方式的容器由AnnotationConfigApplicationContext表示，其实例化方式主要有以下几种：

一、对于只有一个**@Configuration**注解的配置类，可以使用如下方式初始化容器：

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applicati
```

二、对于有多个**@Configuration**注解的配置类，可以使用如下方式初始化容器：

```
AnnotationConfigApplicationContext ctx1 = new AnnotationConfigApplicationContext(Applicat
```

或者

```
AnnotationConfigApplicationContext ctx2 = new AnnotationConfigApplicationContext();
ctx2.register(ApplicationContextConfig.class);
ctx2.register(ApplicationContextConfig2.class);
```

三、对于【**12.3** 注解实现**Bean**定义】中通过扫描类路径中的特殊注解类来自动注册**Bean**定义，可以使用如下方式来实现：

```
public void testComponentScan() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("cn.javass.chapter12.confiuration");
    ctx.refresh();
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

以上配置方式等价于基于XML方式中的如下配置：

```
<context:component-scan base-package="cn.javass.chapter12.confiuration"/>
```

四、在**web**环境中使用基于**Java**方式的配置，通过修改通用配置实现，详见【**10.1.2**通用配置】：

1、修改通用配置中的Web应用上下文实现，在此需要使用AnnotationConfigWebApplicationContext：

```
<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
</context-param>
```

2、指定加载配置类，类似于指定加载文件位置，在基于Java方式中需要指定需要加载的配置类：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig,
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig2
    </param-value>
</context-param>
```

- **contextConfigLocation**：除了可以指定配置类，还可以指定“扫描的类路径”，其加载步骤如下：

1、首先验证指定的配置是否是类，如果是则通过注册配置类来完成Bean定义加载，即如通过ctx.register(ApplicationContextConfig.class)加载定义；

2、如果指定的配置不是类，则通过扫描类路径方式加载注解Bean定义，即将通过ctx.scan("cn.javass.chapter12.confiuration")加载Bean定义。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/2550.html】

【第十二章】零配置 之 12.5 综合示例-积分商城——跟我学spring3

12.5 综合示例

12.5.1 概述

在第十一章中我们介绍了SSH集成，在进行SSH集成时都是通过基于XML配置文件配置每层的Bean，从而产生许多XML配置文件，本节将通过注解方式消除部分XML配置文件，实现所谓的零配置。

12.5.2 项目拷贝

1、拷贝【第十一章 SSH集成开发】中的“pointShop”项目将其命名为“pointShop2”；

2、修改“pointShop2”项目下的“.settings”文件夹下的“org.eclipse.wst.common.component”文件，将“**<property name="context-root" value="pointShop"/>**”修改为“**<property name="context-root" value="pointShop2"/>**”，即该web项目的上下文为“pointShop2”，在浏览器中可以通过http://localhost:8080/pointShop2来访问该web项目。

12.5.3 数据访问层变化

将dao层配置文件中的dao实现Bean定义删除，通过在dao实现类头上添加“@Repository”来定义dao实现Bean，并通过注解@Autowired来完成依赖注入。

1、删除DAO层配置文件(cn/javass/point/dao/applicationContext-hibernate.xml)中的如下配置：

```
<bean id="abstractDao" abstract="true" init-method="init">
<property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="goodsDao" class="cn.javass.point.dao.hibernate.GoodsHibernateDao"
parent="abstractDao"/>
<bean id="goodsCodeDao" class="cn.javass.point.dao.hibernate.GoodsCodeHibernateDao"
parent="abstractDao"/>
```

2、修改通用DAO实现cn.javass.commons.dao.hibernate.BaseHibernateDao，通过注解实现依赖注入和指定初始化方法：

```
public abstract class BaseHibernateDao<M extends Serializable, PK extends Serializable> e
    //省略类字段
    @Autowired @Required
    public void setSf(SessionFactory sf) {
        setSessionFactory(sf);
    }
    @PostConstruct
    @SuppressWarnings("unchecked")
    public void init() {
        //省略具体实现代码
    }
}
```

- setSf方法：通过@Autowired注解自动注入SessionFactory实现；
- init方法：通过@PostConstruct注解表示该方法是初始化方法；

3、修改cn.javass.point.dao.hibernate.GoodsHibernateDao，在该类上添加@Repository注解来进行DAO层Bean定义：

```
@Repository
public class GoodsHibernateDao extends BaseHibernateDao<GoodsModel, Integer> implements I
……
}
```

4、修改cn.javass.point.dao.hibernate.GoodsCodeHibernateDao，在该类上添加@Repository注解来进行DAO层Bean定义：

```
@Repository
public class GoodsCodeHibernateDao extends BaseHibernateDao<GoodsCodeModel, Integer> impl
……
}
```

DAO层到此就修改完毕，其他地方无需修改。

12.5.4 业务逻辑层变化

将service层配置文件中的service实现Bean定义删除，通过在service实现类头上添加“@Service”来定义service实现Bean，并通过注解@Autowired来完成依赖注入。

1、删除Service层配置文件(cn/javass/point/service/applicationContext-service.xml)中的如下配置：

```
<bean id="goodsService" class="cn.javass.point.service.impl.GoodsServiceImpl">
    <property name="dao" ref="goodsDao"/>
</bean>
<bean id="goodsCodeService" class="cn.javass.point.service.impl.GoodsCodeServiceImpl">
    <property name="dao" ref="goodsCodeDao"/>
    <property name="goodsService" ref="goodsService"/>
</bean>
```

2、修改cn.javass.point.service.impl.GoodsServiceImpl，在该类上添加@Service注解来进行Service层Bean定义：

```
@Service
public class GoodsServiceImpl extends BaseServiceImpl<GoodsModel, Integer> implements IGo

    @Autowired @Required
    public void setGoodsDao(IGoodsDao dao) {
        setDao(dao);
    }
}
```

- **setGoodsDao**方法：用于注入IGoodsDao实现，此处直接委托给setDao方法。

3、修改cn.javass.point.service.impl.GoodsCodeServiceImpl，在该类上添加@Service注解来进行Service层Bean定义：

```
@Service
public class GoodsCodeServiceImpl extends BaseServiceImpl<GoodsCodeModel, Integer> implem
    @Autowired @Required
    public void setGoodsCodeDao(IGoodsCodeDao dao) {
        setDao(dao);
    }
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsCodeDao**方法：用于注入IGoodsCodeDao实现，此处直接委托给setDao方法；
- **setGoodsService**方法：用于注入IGoodsService实现。

Service层到此就修改完毕，其他地方无需修改。

12.5.5 表现层变化

类似于数据访问层和业务逻辑层修改，对于表现层配置文件直接删除，通过在action实现类头上添加"@Controller"来定义action实现Bean，并通过注解@Autowired来完成依赖注入。

1、 删除表现层所有Spring配置文件(cn/javass/point/web)：

```
cn/javass/point/web/pointShop-admin-servlet.xml
cn/javass/point/web/pointShop-front-servlet.xml
```

2、修改表现层管理模块的cn.javass.point.web.admin.action.GoodsAction，在该类上添加@Controller注解来进行表现层Bean定义，且作用域为"prototype"：

```
@Controller("/admin/goodsAction")
@Scope("prototype")
public class GoodsAction extends BaseAction {
    private IGoodsService goodsService;
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsService**方法：用于注入IGoodsService实现。

3、修改表现层管理模块的cn.javass.point.web.admin.action.GoodsCodeAction，在该类上添加@Controller注解来进行表现层Bean定义，且作用域为“prototype”：

```
@Controller("/admin/goodsCodeAction")
@Scope("prototype")
public class GoodsCodeAction extends BaseAction {
    @Autowired @Required
    public void setGoodsCodeService(IGoodsCodeService goodsCodeService) {
        this.goodsCodeService = goodsCodeService;
    }
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsCodeService**方法：用于注入IGoodsCodeService实现；
- **setGoodsService**方法：用于注入IGoodsService实现。

3、修改表现层前台模块的cn.javass.point.web.front.action.GoodsAction，在该类上添加@Controller注解来进行表现层Bean定义，且作用域为“prototype”：

```
@Controller("/front/goodsAction")
@Scope("prototype")
public class GoodsAction extends BaseAction {
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
    @Autowired @Required
    public void setGoodsCodeService(IGoodsCodeService goodsCodeService) {
        this.goodsCodeService = goodsCodeService;
    }
}
```

- **setGoodsCodeService**方法：用于注入IGoodsCodeService实现；
- **setGoodsService**方法：用于注入IGoodsService实现。

12.5.6 其他变化

1、定义一个基于Java方法的配置类，用于加载XML配置文件：

```
package cn.javass.point;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
@Configuration
@ImportResource(
        {"classpath:applicationContext-resources.xml",
         "classpath:cn/javass/point/dao/applicationContext-hibernate.xml",
         "classpath:cn/javass/point/service/applicationContext-service.xml"
        })
public class AppConfig {
}
```

该类用于加载零配置中一般不变的XML配置文件，如事务管理，数据源、SessionFactory，这些在几乎所有项目中都是类似的，因此可以作为通用配置。

2、修改集成其它Web框架的通用配置，将如下配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:applicationContext-resources.xml,
        classpath:cn/javass/point/dao/applicationContext-hibernate.xml,
        classpath:cn/javass/point/service/applicationContext-service.xml,
        classpath:cn/javass/point/web/pointShop-admin-servlet.xml,
        classpath:cn/javass/point/web/pointShop-front-servlet.xml
    </param-value>
</context-param>
```

修改为如下配置：

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
       org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>cn.javass.point</param-value>
</context-param>
```

- **contextClass**：使用notationConfigWebApplicationContext替换默认的XmlWebApplicationContext；
- **contextConfigLocation**：指定为“cn.javass.point”，表示将通过扫描该类路径“cn.javass.point”下的注解类来进行加载Bean定义。

启动pointShop2项目，在浏览器输入http://localhost:8080/pointShop2/admin/goods/list.action访问积分商城后台，如果没问题说明零配置整合成功。

到此零配置方式实现SSH集成已经整合完毕，相对于基于XML方式主要减少了配置的数量和配置文件的数量。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/2553.html】

【第十三章】 测试 之 13.1 概述 13.2 单元测试 ——跟我学spring3

13.1 概述

13.1.1 测试

软件测试的目的首先是为了保证软件功能的正确性，其次是为了保证软件的质量，软件测试相当复杂，已经超出本书所涉及的范围，本节将只介绍软件测试流程中前两个步骤：单元测试和集成测试。

Spring提供了专门的测试模块用于简化单元测试和集成测试，单元测试和集成测试一般由程序员实现。

13.2 单元测试

13.2.1 概述

单元测试是最细粒度的测试，即具有原子性，通常测试的是某个功能（如测试类中的某个方法的功能）。

采用依赖注入后我们的代码对Spring IoC容器几乎没有任何依赖，因此在对我们代码进行测试时无需依赖Spring IoC容器，我们只需要通过简单的实例化对象、注入依赖然后测试相应方法来测试该方法是否完成我们预期的功能。

在本书中使用的传统开发流程，即先编写代码实现功能，然后再写测试来验证功能是否正确，而不是测试驱动开发，测试驱动开发是指在编写代码实现功能之前先写测试，然后再根据测试来写满足测试要求的功能代码，通过测试来驱动开发，如果对测试驱动开发感兴趣推荐阅读【测试驱动开发的艺术】。

在实际工作中，应该只对一些复杂的功能进行单元测试，对于一些简单的功能（如数据访问层的CRUD）没有必要花费时间进行单元测试。

Spring对单元测试提供如下支持：

- Mock对象：Spring通过Mock对象来简化一些场景的单元测试：

JNDI测试支持：在org.springframework.mock.jndi包下通过了SimpleNamingContextBuilder来来创建JNDI上下文Mock对象，从而无需依赖特定Java EE容器即可完成JNDI测试。

web测试支持：在org.springframework.mock.web包中提供了一组Servlet API的Mock对象，从而可以无需Web容器即可测试web层的类。

- 工具类：通过通用的工具类来简化编写测试代码：

反射工具类：在org.springframework.test.util包下的ReflectionTestUtils能通过反射完成类的非public字段或setter方法的调用；

JDBC工具类：在org.springframework.test.util包下的SimpleJdbcTestUtils能读取一个sql脚本文件并执行来简化SQL的执行，还提供了如清空表、统计表中行数的简便方法来简化测试代码的编写。

接下来让我们学习一下开发过程中各层代码如何编写测试用例。

13.2.2 准备测试环境

1、**Junit**安装：将Junit 4包添加到"pointShop"项目中，具体方法请参照【2.2.3 Hello World】。

2、**jMock**安装：到jMock官网【http://www.jmock.org/】下载最新的jMock包，在本书中使用jMock2.5.1版本，将下载的"jmock-2.5.1-jars.zip "包中的如下jar包拷贝到项目的lib目录下并添加到类路径：

- objenesis-1.0.jar
- jmock-script-2.5.1.jar
- jmock-legacy-2.5.1.jar
- jmock-junit4-2.5.1.jar
- jmock-junit3-2.5.1.jar
- jmock-2.5.1.jar
- hamcrest-library-1.1.jar
- hamcrest-core-1.1.jar
- bsh-core-2.0b4.jar

注：cglib包无需添加到类路径，因为我们之前已经提供。

3、添加**Spring**测试支持包：将下载的spring-framework-3.0.5.RELEASE-with-docs.zip包中的如下jar包拷贝到项目的lib目录下并添加到类路径：

dist\org.springframework.test-3.0.5.RELEASE.jar

4、在"pointShop"项目下新建test文件夹并将其添加到【Java Build Path】中，该文件夹用于存放测试代码，从而分离测试代码和开发代码。

到此测试环境搭建完毕。

13.2.3 数据访问层

数据访问层单元测试，目的是测试该层定义的接口实现方法的行为是否正确，其实本质是测试是否正确与数据库交互，是否发送并执行了正确的SQL，SQL执行成功后是否正确的组装了业务逻辑层需要的数据。

数据访问层单元测试通过Mock对象与数据库交互的API来完成测试。

接下来让我们学习一下如何进行数据访问层单元测试：

1、在**test**文件夹下创建如下测试类：

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsHibernateDaoUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};
    //2、Mock HibernateTemplate类
    private final HibernateTemplate mockHibernateTemplate = context.mock(HibernateTemplat
    private IGoodsDao goodsDao = null;

    @Before
    public void setUp() {
        //3、创建IGoodsDao实现
        GoodsHibernateDao goodsDaoTemp = new GoodsHibernateDao();
        //4、通过ReflectionTestUtils注入需要的非public字段数据
        ReflectionTestUtils.setField(goodsDaoTemp, "entityClass", GoodsModel.class);
        //5、注入mockHibernateTemplate对象
        goodsDaoTemp.setHibernateTemplate(mockHibernateTemplate);
        //6、赋值给我们要使用的接口
        goodsDao = goodsDaoTemp;
}
}
```

- Mockery：jMock核心类，用于创建Mock对象的，通过其mock方法来创建相应接口或类的Mock对象。
- goodsDaoTemp：需要测试的IGoodsDao实现，通过ReflectionTestUtils注入需要的非public字段数据。

2、测试支持写完后，接下来测试一下**IGoodsDao**的**get**方法是否满足需求：

```
@Test
public void testSave () {
    //7、创建需要的Model数据
    final GoodsModel expected = new GoodsModel();
    //8、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
        {
            //9、表示需要调用且只调用一次mockHibernateTemplate的get方法，
            //且get方法参数为(GoodsModel.class, 1)，并将返回goods
            one(mockHibernateTemplate).get(GoodsModel.class, 1);
            will(returnValue(expected));
         }
    });
    //10、调用goodsDao的get方法，在内部实现中将委托给
    //getHibernateTemplate().get(this.entityClass, id);
    //因此按照第8步定义的预期行为将返回goods
    GoodsModel actual = goodsDao.get(1);
    //11、来验证第8步定义的预期行为是否调用了
   context.assertIsSatisfied();
    //12、验证goodsDao.get(1)返回结果是否正确
    Assert.assertEquals(goods, expected);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第9步定义了需要调用一次且只调用一次mockHibernateTemplate的get方法，且get方法参数为(GoodsModel.class, 1)，并将返回goods对象。
- **goodsDao.get(1)**：调用goodsDao的get方法，在内部实现中将委托给“getHibernateTemplate().get(this.entityClass, id)”。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(expected, actual)**：用于验证“goodsDao.get(1) ”返回的结果是否是预期结果。

以上测试方法其实是没有必要的，对于非常简单的CRUD没有必要写单元测试，只有相当复杂的方法才有必要写单元测试。

这种通过Mock对象来测试数据访问层代码其实一点意义没有，因为这里没有与数据库交互，无法验证真实环境中与数据库交互是否正确，因此这里只是告诉你如何测试数据访问层代码，在实际工作中一般通过集成测试来完成数据访问层测试。

13.2.4 业务逻辑层

业务逻辑单元测试，目的是测试该层的业务逻辑是否正确并通过Mock 数据访问层对象来隔离与数据库交互，从而无需连接数据库即可测试业务逻辑是否正确。

接下来让我们学习一下如何进行业务逻辑层单元测试：

1、在**test**文件夹下创建如下测试类：

```
package cn.javass.point.service.impl;
//省略import
public class GoodsCodeServiceImplUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    //2、Mock IGoodsCodeDao接口
    private IGoodsCodeDao goodsCodeDao = context.mock(IGoodsCodeDao.class);;

    private IGoodsCodeService goodsCodeService;

    @Before
    public void setUp() {
        GoodsCodeServiceImpl goodsCodeServiceTemp = new GoodsCodeServiceImpl();
        //3、依赖注入
        goodsCodeServiceTemp.setDao(goodsCodeDao);
        goodsCodeService = goodsCodeServiceTemp;
}
}
```

以上测试支持代码和数据访问层测试代码非常类似，在此不再阐述。

2、测试支持写完后，接下来测试一下购买商品**Code**码是否满足需求：

测试业务逻辑时需要分别测试多种场景，即如在某种场景下成功或失败等等，即测试应该全面，每个功能点都应该测试到。

2.1、测试购买失败的场景：

```
@Test(expected = NotCodeException.class)
public void testBuyFail() {
    final int goodsId = 1;
    //4、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //5、表示需要调用goodsCodeDao对象的getOneNotExchanged一次且仅以此
          //且返回值为null
          one(goodsCodeDao).getOneNotExchanged(goodsId);
          will(returnValue(null));
      }
    });
    goodsCodeService.buy("test", goodsId);
    context.assertIsSatisfied();
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第5步定义了需要调用一次且只调用一次goodsCodeDao的getOneNotExchanged方法，且getOneNotExchanged方法参数为(goodsId)，并将返回null。
- **goodsCodeService.buy("test", goodsId)**：调用goodsCodeService的buy方法，由于调用goodsCodeDao的getOneNotExchanged方法将返回null，因此buy方法将抛出“NotCodeException”异常，从而表示没有Code码。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- 由于我们在预期行为中调用getOneNotExchanged将返回null，因此测试将失败且抛出NotCodeException异常。

2.2、测试购买成功的场景：

```
@Test()
public void testBuySuccess () {
    final int goodsId = 1;
    final GoodsCodeModel goodsCode = new GoodsCodeModel();
    //6、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //7、表示需要调用goodsCodeDao对象的getOneNotExchanged一次且仅以此
          //且返回值为null
          one(goodsCodeDao).getOneNotExchanged(goodsId);
          will(returnValue(goodsCode));
          //8、表示需要调用goodsCodeDao对象的save方法一次且仅一次
          //且参数为goodsCode
          one(goodsCodeDao).save(goodsCode);
        }
      });
    goodsCodeService.buy("test", goodsId);
    context.assertIsSatisfied();
    Assert.assertEquals(goodsCode.isExchanged(), true);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第7步定义了需要调用一次且只调用一次goodsCodeDao的getOneNotExchanged方法，且getOneNotExchanged方法参数为(goodsId)，并将返回goodsCode对象；第8步定义了需要调用goodsCodeDao对象的save一次且仅一次。
- **goodsCodeService.buy("test", goodsId)**：调用goodsCodeService的buy方法，由于调用goodsCodeDao的getOneNotExchanged方法将返回goodsCode，因此buy方法将成功执行。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(goodsCode.isExchanged(), true)**：表示goodsCode已经被购买过了。
- 由于我们在预期行为中调用getOneNotExchanged将返回一个goodsCode对象，因此测试将成功，如果失败说明业务逻辑写错了。

到此业务逻辑层测试完毕，在进行业务逻辑层测试时我们只关心业务逻辑是否正确，而不关心底层数据访问层如何实现，因此测试业务逻辑层时只需Mock 数据访问层对象，然后定义一些预期行为来满足业务逻辑测试需求即可。

13.2.5 表现层

表现层测试包括如Struts2的Action单元测试、拦截器单元测试、JSP单元测试等等，在此我们只学习Struts2的Action单元测试。

Struts2的Action测试相对业务逻辑层测试相对复杂一些，因为牵扯到使用如Servlet API、ActionContext等等，这里需要通过stub（桩）实现或mock对象来模拟如HttpServletRequest等对象。

一、首先学习一些最简单的Action测试：

1、在**test**文件夹下创建如下测试类：

```
package cn.javass.point.web.front;
import cn.javass.point.service.IGoodsCodeService;
import cn.javass.point.web.front.action.GoodsAction;
//省略部分import
public class GoodsActionUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    //2、Mock IGoodsCodeService接口
    private IGoodsCodeService goodsCodeService = context.mock(IGoodsCodeService.class);

    private GoodsAction goodsAction;

    @Before
    public void setUp() {
        goodsAction = new GoodsAction();
        //3、依赖注入
        goodsAction.setGoodsCodeService(goodsCodeService);
}
}
```

以上测试支持代码和业务逻辑层测试代码非常类似，在此不再阐述。

2、测试支持写完后，接下来测试一下前台购买商品**Code**码是否满足需求：

类似于测试业务逻辑时需要分别测试多种场景，测试Action同样需要分别测试多种场景。

2.1、测试购买失败的场景：

```
@Test
public void testBuyFail() {
    final int goodsId = 1;
    //4、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //5、表示需要调用goodsCodeService对象的buy方法一次且仅一次
          //且抛出NotCodeException异常
          one(goodsCodeService).buy("test", goodsId);
      will(throwException(new NotCodeException()));
      }
    });
    //6、模拟Struts注入请求参数
    goodsAction.setGoodsId(goodsId);
    String actualResultCode = goodsAction.buy();
    context.assertIsSatisfied();
    Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualRe
    Assert.assertTrue(goodsAction.getActionErrors().size() > 0);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第5步定义了需要调用goodsCodeService对象的buy方法一次且仅一次且将抛出NotCodeException异常。
- **goodsAction.setGoodsId(goodsId)**：用于模拟Struts注入请求参数，即完成数据绑定。

- **goodsAction.buy()**：调用goodsAction的buy方法，该方法将委托给IGoodsCodeService实现完成，返回值用于定位视图。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultCode)**：验证返回的Result是否是我们指定的。

- **Assert.assertTrue(goodsAction.getActionErrors().size() > 0)**：表示执行Action时有错误，即Action动作错误。如果条件不成立，说明我们Action功能是错误的，需要修改。

2.2、测试购买成功的场景：

```
@Test
public void testBuySuccess() {
    final int goodsId = 1;
    final GoodsCodeModel goodsCode = new GoodsCodeModel();
    //7、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //8、表示需要调用goodsCodeService对象的buy方法一次且仅一次
          //且返回goodsCode对象
          one(goodsCodeService).buy("test", goodsId);
          will(returnValue(goodsCode));
      }
    });
    //9、模拟Struts注入请求参数
    goodsAction.setGoodsId(goodsId);
    String actualResultCode = goodsAction.buy();
    context.assertIsSatisfied();
    Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualRe
    Assert.assertTrue(goodsAction.getActionErrors().size() == 0);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第5步定义了需要调用goodsCodeService对象的buy方法一次且仅一次且将返回goodsCode对象。
- **goodsAction.setGoodsId(goodsId)**：用于模拟Struts注入请求参数，即完成数据绑定。
- **goodsAction.buy()**：调用goodsAction的buy方法，该方法将委托给IGoodsCodeService实现完成，返回值用于定位视图。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultCode)**：验证返回的Result是否是我们指定的。

- **Assert.assertTrue(goodsAction.getActionErrors().size() == 0)**：表示执行Action时没有错误，即Action动作正确。如果条件不成立，说明我们Action功能是错误的，需要修改。

通过模拟ActionContext对象内容从而可以非常容易的测试Action中各种与http请求相关情况，无需依赖web服务器即可完成测试。但对于如果我们使用htpp请求相关对象的该如何测试？如果我们需要使用ActionContext获取值栈数据应该怎么办？这就需要Struts提供的junit插件支持了。我们会在集成测试中介绍。

对于表现层其他功能的单元测试本书不再介绍，如JSP单元测试、拦截器单元测试等等。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/2555.html】

【第十三章】 测试 之 13.3 集成测试 ——跟我学spring3

13.3 集成测试

13.3.1 概述

集成测试是在单元测试之上，通常是将一个或多个已进行过单元测试的组件组合起来完成的，即集成测试中一般不会出现Mock对象，都是实实在在的真实实现。

对于单元测试，如前边在进行数据访问层单元测试时，通过Mock HibernateTemplate对象然后将其注入到相应的DAO实现，此时单元测试只测试某层的某个功能是否正确，对其他层如何提供服务采用Mock方式提供。

对于集成测试，如要进行数据访问层集成测试时，需要实实在在的HibernateTemplate对象然后将其注入到相应的DAO实现，此时集成测试将不仅测试该层功能是否正确，还将测试服务提供者提供的服务是否正确执行。

使用Spring的一个好处是能非常简单的进行集成测试，无需依赖web服务器或应用服务器即可完成测试。Spring通过提供一套TestContext框架来简化集成测试，使用TestContext测试框架能获得许多好处，如Spring IoC容器缓存、事务管理、依赖注入、Spring测试支持类等等。

13.3.2 Spring TestContext框架支持

Spring TestContext框架提供了一些通用的集成测试支持，主要提供如下支持：

一、上下文管理及缓存：

对于每一个测试用例（测试类）应该只有一个上下文，而不是每个测试方法都创建新的上下文，这样有助于减少启动容器的开销，提供测试效率。可通过如下方式指定要加载的上下文：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
                     "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
public class GoodsHibernateDaoIntegrationTest {
}
```

- **locations**：指定Spring配置文件位置；
- **inheritLocations**：如果设置为false，将屏蔽掉父类中使用该注解指定的配置文件位置，默认为true表示继承父类中使用该注解指定的配置文件位置。

二、**Test Fixture**（测试固件）的依赖注入：

Test Fixture可以指运行测试时需要的任何东西，一般通过@Before定义的初始化Fixture方法准备这些资源，而通过@After定义的销毁Fixture方法销毁或还原这些资源。

Test Fixture的依赖注入就是使用Spring IoC容器的注入功能准备和销毁这些资源。可通过如下方式注入Test Fixture：

```
@Autowired
private IGoodsDao goodsDao;
@Autowired
private ApplicationContext ctx;
```

即可以通过Spring提供的注解实现Bean的依赖注入来完成Test Fixture的依赖注入。

三、事务管理：

开启测试类的事务管理支持，即使用Spring 容器的事务管理功能，从而可以独立于应用服务器完成事务相关功能的测试。为了使测试中的事务管理起作用需要通过如下方式开启测试类事务的支持：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
               "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
@TransactionConfiguration(
transactionManager = "txManager", defaultRollback=true)
public class GoodsHibernateDaoIntegrationTest {
}
```

Spring提供如下事务相关注解来支持事务管理：

- @Transactional：使用@Transactional注解的类或方法将得到事务支持
- **transactionManager:**指定事务管理器；
- **defaultRollback**：是否回滚事务，默认为true表示回滚事务。

Spring还通过提供如下注解来简化事务测试：

- **@Transactional**：使用@Transactional注解的类或方法表示需要事务支持；
- **@NotTransactional**：只能注解方法，使用@NotTransactional注解的方法表示不需要事务支持，即不运行在事务中，Spring 3开始已不推荐使用；
- **@BeforeTransaction**和**@AfterTransaction**：使用这两个注解注解的方法定义了在一个事务性测试方法之前或之后执行的行为，且被注解的方法将运行在该事务性方法的事务之外。
- **@Rollback(true)**：默认为true，用于替换@TransactionConfiguration中定义的defaultRollback指定的回滚行为。

四、常用注解支持：Spring框架提供如下注解来简化集成测试：

- **@DirtiesContext**：表示每个测试方法执行完毕需关闭当前上下文并重建一个全新的上下文，即不缓存上下文。可应用到类或方法级别，但在JUnit 3.8中只能应用到方法级别。
- **@ExpectedException**：表示被注解的方法预期将抛出一个异常，使用如@ExpectedException(NotCodeException.class)来指定异常，定义方式类似于Junit 4中的@Test(expected = NotCodeException.class)，@ExpectedException注解和@Test(expected =……)应该两者选一。
- @Repeat：表示被注解的方法应被重复执行多少次，使用如**@Repeat(2)**方式指定。
- **@Timed**：表示被注解的方法必须在多长时间内运行完毕，超时将抛出异常，使用如@Timed(millis=10)方式指定，单位为毫秒。注意此处指定的时间是如下方法执行时间之和：测试方法执行时间（或者任何测试方法重复执行时间之和）、@Before和@After注解的测试方法之前和之后执行的方法执行时间。而Junit 4中的@Test(timeout=2)指定的超时时间只是测试方法执行时间，不包括任何重复等。
- 除了支持如上注解外，还支持【第十二章 零配置】中依赖注入等注解。

五、**TestContext**框架支持类：提供对测试框架的支持，如Junit、TestNG测试框架，用于集成Spring TestContext和测试框架来简化测试，TestContext框架提供如下支持类：

- **JUnit 3.8**支持类：提供对Spring TestContext框架与Junit3.8测试框架的集成：

AbstractJUnit38SpringContextTests：我们的测试类继承该类后将获取到Test Fixture的依赖注入好处。

AbstractTransactionalJUnit38SpringContextTests：我们的测试类继承该类后除了能得到Test Fixture的依赖注入好处，还额外获取到事务管理支持。

- **JUnit 4.5+**支持类：提供对Spring TestContext框架与Junit4.5+测试框架的集成：

AbstractJUnit4SpringContextTests：我们的测试类继承该类后将获取到Test Fixture的依赖注入好处。

AbstractTransactionalJUnit4SpringContextTests：我们的测试类继承该类后除了能得到Test Fixture的依赖注入好处，还额外获取到事务管理支持。

- 定制 **Junit4.5+**运行器：通过定制自己的Junit4.5+运行器从而无需继承JUnit 4.5+支持类即可完成需要的功能，如Test Fixture的依赖注入、事务管理支持，

@RunWith(SpringJUnit4ClassRunner.class)：使用该注解注解到测试类上表示将集成Spring TestContext和Junit 4.5+测试框架。

@TestExecutionListeners：该注解用于指定TestContext框架的监听器用于与TestContext框架管理器发布的测试执行事件进行交互，TestContext框架提供如下三个默认的监听器：DependencyInjectionTestExecutionListener、DirtiesContextTestExecutionListener、TransactionalTestExecutionListener分别完成对Test Fixture的依赖注入、@DirtiesContext支持和事务管理支持，即在默认情况下将自动注册这三个监听器，另外还可以使用如下方式指定监听器：

```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class GoodsHibernateDaoIntegrationTest {
}
```

如上配置将通过定制的Junit4.5+运行器运行，但不会完成Test Fixture的依赖注入、事务管理等等，如果只需要Test Fixture的依赖注入，可以使用@TestExecutionListeners({DependencyInjectionTestExecutionListener.class})指定。

- **TestNG**支持类：提供对Spring TestContext框架与TestNG测试框架的集成：

AbstractTestNGSpringContextTests：我们的测试类继承该类后将获取到Test Fixture的依赖注入好处。

AbstractTransactionalTestNGSpringContextTests：我们的测试类继承该类后除了能得到Test Fixture的依赖注入好处，还额外获取到事务管理支持。

到此Spring TestContext测试框架减少完毕了，接下来让我们学习一下如何进行集成测试吧。

13.3.3 准备集成测试环境

对于集成测试环境各种配置应该和开发环境或实际生产环境配置相分离，即集成测试时应该使用单独搭建一套独立的测试环境，不应使用开发环境或实际生产环境的配置，从而保证测试环境、开发、生产环境相分离。

1、拷贝一份Spring资源配置文件applicationContext-resources.xml，并命名为applicationContext-resources-test.xml表示用于集成测试使用，并修改如下内容：

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
        <list>
            <value>classpath:resources-test.properties</value>
        </list>
  </property>
</bean>
```

2、拷贝一份替换配置元数据的资源文件（**resources/resources.properties**），并命名为**resources-test.properties**表示用于集成测试使用，并修改为以下内容：

```
db.driver.class=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:mem:point_shop
db.username=sa
db.password=
#Hibernate属性
hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.hbm2ddl.auto=create-drop
hibernate.show_sql=false
hibernate.format_sql=true
```

- **jdbc:hsqldb:mem:point_shop**：我们在集成测试时将使用HSQLDB，并采用内存数据

库模式运行；

- **hibernate.hbm2ddl.auto=create-drop**：表示在创建SessionFactory时根据Hibernate映射配置创建相应Model的表结构，并在SessionFactory关闭时删除这些表结构。

到此我们测试环境修改完毕，在进行集成测试时一定要保证测试环境、开发环境、实际生产环境相分离，即对于不同的环境使用不同的配置文件。

13.3.4 数据访问层

数据访问层集成测试，同单元测试一样目的不仅测试该层定义的接口实现方法的行为是否正确，而且还要测试是否正确与数据库交互，是否发送并执行了正确的SQL，SQL执行成功后是否正确的组装了业务逻辑层需要的数据。

数据访问层集成测试不再通过Mock对象与数据库交互的API来完成测试，而是使用实实在在存在的与数据库交互的对象来完成测试。

接下来让我们学习一下如何进行数据访问层集成测试：

1、在**test**文件夹下创建如下测试类：

```
package cn.javass.point.dao.hibernate;
//省略import
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
                  "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
@TransactionConfiguration(transactionManager = "txManager", defaultRollback=false)
public class GoodsHibernateDaoIntegrationTest {
    @Autowired
    private ApplicationContext ctx;
    @Autowired
    private IGoodsCodeDao goodsCodeDao;
}
```

- **@RunWith(SpringJUnit4ClassRunner.class)**：表示使用自己定制的Junit4.5+运行器来运行测试，即完成Spring TestContext框架与Junit集成；
- **@ContextConfiguration**：指定要加载的Spring配置文件，此处注意我们的Spring资源配置文件为“applicationContext-resources-test.xml”；
- **@TransactionConfiguration**：开启测试类的事务管理支持配置，并指定事务管理器和默认回滚行为；
- **@Autowired**：完成Test Fixture（测试固件）的依赖注入。

2、测试支持写完后，接下来测试一下分页查询所有已发布的商品是否满足需求：

```
@Transactional
@Rollback
@Test
public void testListAllPublishedSuccess() {
    GoodsModel goods = new GoodsModel();
    goods.setDeleted(false);
    goods.setDescription("");
    goods.setName("测试商品");
    goods.setPublished(true);
    goodsDao.save(goods);
    Assert.assertTrue(goodsDao.listAllPublished(1).size() == 1);
    Assert.assertTrue(goodsDao.listAllPublished(2).size() == 0);
}
```

- **@Transactional**：表示测试方法将允许在事务环境；
- **@Rollback**：表示替换@ContextConfiguration指定的默认事务回滚行为，即将在测试方法执行完毕时回滚事务。

数据访问层的集成测试也是非常简单，与数据访问层的单元测试类似，也应该只对复杂的数据访问层代码进行测试。

13.3.5 业务逻辑层

业务逻辑层集成测试，目的同样是测试该层的业务逻辑是否正确，对于数据访问层实现通过Spring IoC容器完成装配，即使用真实的数据访问层实现来获取相应的底层数据。

接下来让我们学习一下如何进行业务逻辑层集成测试：

1、在**test**文件夹下创建如下测试类：

```
@ContextConfiguration(
locations={"classpath:applicationContext-resources-test.xml",
             "classpath:cn/javass/point/dao/applicationContext-hibernate.xml",
             "classpath:cn/javass/point/service/applicationContext-service.xml"})
@TransactionConfiguration(transactionManager = "txManager", defaultRollback=false)
public class GoodsCodeServiceImplIntegrationTest extends AbstractJUnit4SpringContextTests
    @Autowired
    private IGoodsCodeService goodsCodeService;
    @Autowired
    private IGoodsService goodsService;
}
```

- **AbstractJUnit4SpringContextTests**：表示将Spring TestContext框架与Junit4.5+测试框架集成；
- **@ContextConfiguration**：指定要加载的Spring配置文件，此处注意我们的Spring资源配置文件为“applicationContext-resources-test.xml”；
- **@TransactionConfiguration**：开启测试类的事务管理支持配置，并指定事务管理器和默认回滚行为；
- **@Autowired**：完成Test Fixture（测试固件）的依赖注入。

2、测试支持写完后，接下来测试一下购买商品**Code**码是否满足需求：

2.1、测试购买失败的场景：

```
@Transactional
@Rollback
@ExpectedException(NotCodeException.class)
@Test
public void testBuyFail() {
    goodsCodeService.buy("test", 1);
}
```

由于我们数据库中没有相应商品的Code码，因此将抛出NotCodeException异常。

2.2、测试购买成功的场景：

```
@Transactional
@Rollback
@Test
public void testBuySuccess() {
    //1.添加商品
    GoodsModel goods = new GoodsModel();
    goods.setDeleted(false);
    goods.setDescription("");
    goods.setName("测试商品");
    goods.setPublished(true);
    goodsService.save(goods);

    //2.添加商品Code码
    GoodsCodeModel goodsCode = new GoodsCodeModel();
    goodsCode.setGoods(goods);
    goodsCode.setCode("test");
    goodsCodeService.save(goodsCode);
    //3.测试购买商品Code码
    GoodsCodeModel resultGoodsCode = goodsCodeService.buy("test", 1);
    Assert.assertEquals(goodsCode.getId(), resultGoodsCode.getId());
}
```

由于我们添加了指定商品的Code码因此购买将成功，如果失败说明业务写错了，应该重写。

业务逻辑层的集成测试也是非常简单，与业务逻辑层的单元测试类似，也应该只对复杂的业务逻辑层代码进行测试。

13.3.5 表现层

对于表现层集成测试，同样类似于单元测试，但对于业务逻辑层都将使用真实的实现，而不再是通过Mock对象来测试，这也是集成测试和单元测试的区别。

接下来让我们学习一下如何进行表现层Action集成测试：

1、准备**Struts**提供的**junit**插件, 到**struts-2.2.1.1.zip**中拷贝如下**jar**包到类路径：

```
lib\struts2-junit-plugin-2.2.1.1.jar
```

2、测试支持类：Struts2提供StrutsSpringTestCase测试支持类，我们所有的Action测试类都需要继承该类；

3、准备**Spring**配置文件：由于我们的测试类继承StrutsSpringTestCase且将通过覆盖该类的getContextLocations方法来指定Spring配置文件，但由于getContextLocations方法只能返回一个配置文件，因此我们需要新建一个用于导入其他Spring配置文件的配置文件applicationContext-test.xml，具体内容如下：

```
<import resource="classpath:applicationContext-resources-test.xml"/>
<import resource="classpath:cn/javass/point/dao/applicationContext-hibernate.xml"/>
<import resource="classpath:cn/javass/point/service/applicationContext-service.xml"/>
<import resource="classpath:cn/javass/point/web/pointShop-admin-servlet.xml"/>
<import resource="classpath:cn/javass/point/web/pointShop-front-servlet.xml"/>
```

3、在**test**文件夹下创建如下测试类：

```
package cn.javass.point.web.front;
//省略import
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class GoodsActionIntegrationTest extends StrutsSpringTestCase {
    @Override
    protected String getContextLocations() {
        return "classpath:applicationContext-test.xml";
    }
    @Before
    public void setUp() throws Exception {
        //1 指定Struts2配置文件
        //该方式等价于通过web.xml中的<init-param>方式指定参数
        Map<String, String> dispatcherInitParams = new HashMap<String, String>();
        ReflectionTestUtils.setField(this, "dispatcherInitParams", dispatcherInitParams);
        //1.1 指定Struts配置文件位置
        dispatcherInitParams.put("config", "struts-default.xml,struts-plugin.xml,struts.x
        super.setUp();
    }
    @After
    public void tearDown() throws Exception {
        super.tearDown();
    }
}
```

- **@RunWith(SpringJUnit4ClassRunner.class)**：表示使用自己定制的Junit4.5+运行器来运行测试，即完成Spring TestContext框架与Junit集成；
- **@TestExecutionListeners({})**：没有指定任何监听器，即不会自动完成对Test Fixture的依赖注入、@DirtiesContext支持和事务管理支持；
- **StrutsSpringTestCase**：集成测试Struts2+Spring时所有集成测试类必须继承该类；
- **setUp**方法：在每个测试方法之前都执行的初始化方法，其中dispatcherInitParams用于指定等价于在web.xml中的<init-param>方式指定的参数；必须调用super.setUp()用于初始化Struts2和Spring环境。
- **tearDown()**：在每个测试方法之前都执行的销毁方法，必须调用super.tearDown()来销毁Spring容器等。

4、测试支持写完后，接下来测试一下前台购买商品**Code**码是否满足需求：

4.1、测试购买失败的场景：

```
@Test
public void testBuyFail() throws UnsupportedEncodingException, ServletException {
    //2 前台购买商品失败
    //2.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(Integer.MIN_VALUE));
    //2.2 调用前台GoodsAction的buy方法完成购买相应商品的Code码
    executeAction("/goods/buy.action");
    GoodsAction frontGoodsAction = (GoodsAction) ActionContext.getContext().getActionInvo
    //2.3 验证前台GoodsAction的buy方法有错误
    Assert.assertTrue(frontGoodsAction.getActionErrors().size() > 0);
}
```

- **initServletMockObjects()**：用于重置所有http相关对象，如request等；
- **request.setParameter("goodsId", String.valueOf(Integer.MIN_VALUE))**：用于准备请求参数；
- **executeAction("/goods/buy.action")**：通过模拟http请求来调用前台GoodsAction的buy方法完成商品购买
- **Assert.assertTrue(frontGoodsAction.getActionErrors().size() > 0)**：表示执行Action时有错误，即Action动作错误。如果条件不成立，说明我们Action功能是错误的，需要修改。

4.2、测试购买成功的场景：

```
@Test
public void testBuySuccess() throws UnsupportedEncodingException, ServletException {
    //3 后台新增商品
    //3.1 准备请求参数
    request.setParameter("goods.name", "测试商品");
    request.setParameter("goods.description", "测试商品描述");
    request.setParameter("goods.originalPoint", "1");
    request.setParameter("goods.nowPoint", "2");
    request.setParameter("goods.published", "true");
    //3.2 调用后台GoodsAction的add方法完成新增
    executeAction("/admin/goods/add.action");
    //2.3 获取GoodsAction的goods属性
    GoodsModel goods = (GoodsModel) findValueAfterExecute("goods");
    //4 后台新增商品Code码
    //4.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(goods.getId()));
    request.setParameter("codes", "a\rb");
    //4.2 调用后台GoodsCodeAction的add方法完成新增商品Code码
    executeAction("/admin/goodsCode/add.action");
    //5 前台购买商品成功
    //5.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(goods.getId()));
    //5.2 调用前台GoodsAction的buy方法完成购买相应商品的Code码
    executeAction("/goods/buy.action");
    GoodsAction frontGoodsAction = (GoodsAction) ActionContext.getContext().getActionInvo
    //5.3 验证前台GoodsAction的buy方法没有错误
    Assert.assertTrue(frontGoodsAction.getActionErrors().size() == 0);
}
```

- **executeAction("/admin/goods/add.action")**：调用后台GoodsAction的add方法，用于新增商品；

- **executeAction("/admin/goodsCode/add.action")**：调用后台GoodCodeAction的add方法用于新增商品Code码；
- **executeAction("/goods/buy.action")**：调用前台GoodsAction的buy方法，用于购买相应商品，其中Assert.assertTrue(frontGoodsAction.getActionErrors().size() == 0)表示购买成功，即Action动作正确。

表现层Action集成测试介绍就到此为止，如何深入StrutsSpringTestCase来完成集成测试已超出本书范围，如果读者对这部分感兴趣可以到Struts2官网学习最新的测试技巧。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/2557.html】

跟我学 Spring MVC

> 作者：开涛
>
> 来源：跟开涛学SpringMVC

SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结

下载地址

一 开发环境

1、动态web工程

2、部分依赖

```
hibernate-release-4.1.0.Final.zip
hibernate-validator-4.2.0.Final.jar
spring-framework-3.1.1.RELEASE-with-docs.zip
proxool-0.9.1.jar
log4j 1.2.16
slf4j -1.6.1
mysql-connector-java-5.1.10.jar
hamcrest 1.3.0RC2
ehcache 2.4.3
```

3、为了方便学习，暂没有使用maven构建工程

二 工程主要包括内容

1、springMVC + spring3.1.1 + hibernate4.1.0集成

2、通用DAO层 和 Service层

3、二级缓存 Ehcache

4、REST风格的表现层

5、通用分页（两个版本）

5.1、首页 上一页,下一页 尾页 跳转

5.2、上一页 1 2 3 4 5 下一页

6、数据库连接池采用proxool

7、spring集成测试

8、表现层的 java validator框架验证（采用hibernate-validator-4.2.0实现）

9、视图采用JSP，并进行组件化分离

三 TODO LIST 将本项目做成脚手架方便以后新项目查询

1、Service层进行AOP缓存（缓存使用Memcached实现）

2、单元测试（把常见的桩测试、伪实现、模拟对象演示一遍 区别集成测试）

3、监控功能

后台查询hibernate二级缓存 hit/miss率功能

后台查询当前服务器状态功能（如 线程信息、服务器相关信息）

4、spring RPC功能

5、spring集成 quartz 进行任务调度

6、spring集成 java mail进行邮件发送

7、DAO层将各种常用框架集成进来（方便查询）

8、把工作中经常用的东西 融合进去，作为脚手架，方便以后查询

四 集成重点及常见问题

1、spring-config.xml 配置文件：

1.1、该配置文件只加载除表现层之外的所有bean，因此需要如下配置：

```
    <context:component-scan base-package="cn.javass">
        <context:exclude-filter type="annotation" expression="org.springframework.stereot
    </context:component-scan>
```

通过exclude-filter 把所有 @Controller注解的表现层控制器组件排除

1.2、国际化消息文件配置

```
<!-- 国际化的消息资源文件 -->
    <bean id="messageSource" class="org.springframework.context.support.ReloadableResourc
        <property name="basenames">
            <list>
                <!-- 在web环境中一定要定位到classpath 否则默认到当前web应用下找  -->
                <value>classpath:messages</value>
            </list>
        </property>
        <property name="defaultEncoding" value="UTF-8"/>
        <property name="cacheSeconds" value="60"/>
    </bean>
```

此处basenames内一定是 classpath:messages ，如果你写出“messages”，将会到你的web应用的根下找 即你的messages.properties一定在 web应用/messages.propertis。

1.3、hibernate的sessionFactory配置 需要使用org.springframework.orm.hibernate4.LocalSessionFactoryBean，其他都是类似的，具体看源代码。

1・4、<aop:aspectj-autoproxy expose-proxy="true"/> 实现@AspectJ注解的，默认使用AnnotationAwareAspectJAutoProxyCreator进行AOP代理，它是BeanPostProcessor的子类，在容器启动时Bean初始化开始和结束时调用进行AOP代理的创建，因此只对当容器启动时有效，使用时注意此处。

1.5、声明式容器管理事务

建议使用声明式容器管理事务，而不建议使用注解容器管理事务（虽然简单），但太分布式了，采用声明式容器管理事务一般只对service层进行处理。

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" />
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="create*" propagation="REQUIRED" />
        <tx:method name="insert*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="merge*" propagation="REQUIRED" />
        <tx:method name="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="put*" propagation="REQUIRED" />
        <tx:method name="use*" propagation="REQUIRED"/>
        <!--hibernate4必须配置为开启事务 否则 getCurrentSession()获取不到-->
        <tx:method name="get*" propagation="REQUIRED" read-only="true" />
        <tx:method name="count*" propagation="REQUIRED" read-only="true" />
        <tx:method name="find*" propagation="REQUIRED" read-only="true" />
        <tx:method name="list*" propagation="REQUIRED" read-only="true" />
        <tx:method name="*" read-only="true" />
    </tx:attributes>
</tx:advice>
<aop:config expose-proxy="true">
    <!-- 只对业务逻辑层实施事务 -->
    <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service..*.*(..)
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>
```

此处一定注意 使用 hibernate4，在不使用OpenSessionInView模式时，在使用getCurrentSession()时会有如下问题：

当有一个方法list 传播行为为Supports，当在另一个方法getPage()（无事务）调用list方法时会抛出org.hibernate.HibernateException: No Session found for current thread 异常。

这是因为getCurrentSession()在没有session的情况下不会自动创建一个，不知道这是不是Spring3.1实现的bug，欢迎大家讨论下。

因此最好的解决方案是使用REQUIRED的传播行为。

二、spring-servlet.xml：

2.1、表现层配置文件，只应加装表现层Bean，否则可能引起问题。

```
<!-- 开启controller注解支持 -->
<!-- 注：如果base-package=cn.javass 则注解事务不起作用-->
<context:component-scan base-package="cn.javass.demo.web.controller">
    <context:include-filter type="annotation" expression="org.springframework.stereot
</context:component-scan>
```

此处只应该加载表现层组件，如果此处还加载dao层或service层的bean会将之前容器加载的替换掉，而且此处不会进行AOP织入，所以会造成AOP失效问题（如事务不起作用），再回头看我们的1.4讨论的。

2.2、<mvc:view-controller path=*"/"* view-name=*"forward:/index"*/> 表示当访问主页时自动转发到index控制器。

2.3、静态资源映射

```
<!-- 当在web.xml 中   DispatcherServlet使用     <url-pattern>/</url-pattern> 映射时，能映
<mvc:default-servlet-handler/>
<!-- 静态资源映射 -->
<mvc:resources mapping="/images/**" location="/WEB-INF/images/" />
<mvc:resources mapping="/css/**" location="/WEB-INF/css/" />
<mvc:resources mapping="/js/**" location="/WEB-INF/js/" />
```

以上是配置文件部分，接下来来看具体代码。

三、通用DAO层Hibernate4实现

为了减少各模块实现的代码量，实际工作时都会有通用DAO层实现，以下是部分核心代码：

```
public abstract class BaseHibernateDao<M extends java.io.Serializable, PK extends java.io

    protected static final Logger LOGGER = LoggerFactory.getLogger(BaseHibernateDao.class

    private final Class<M> entityClass;
    private final String HQL_LIST_ALL;
    private final String HQL_COUNT_ALL;
    private final String HQL_OPTIMIZE_PRE_LIST_ALL;
    private final String HQL_OPTIMIZE_NEXT_LIST_ALL;
    private String pkName = null;

    @SuppressWarnings("unchecked")
    public BaseHibernateDao() {
        this.entityClass = (Class<M>) ((ParameterizedType) getClass().getGenericSuperclas
        Field[] fields = this.entityClass.getDeclaredFields();
        for(Field f : fields) {
            if(f.isAnnotationPresent(Id.class)) {
                this.pkName = f.getName();
            }
        }

        Assert.notNull(pkName);
        //TODO @Entity name not null
        HQL_LIST_ALL = "from " + this.entityClass.getSimpleName() + " order by " + pkName
        HQL_OPTIMIZE_PRE_LIST_ALL = "from " + this.entityClass.getSimpleName() + " where 
        HQL_OPTIMIZE_NEXT_LIST_ALL = "from " + this.entityClass.getSimpleName() + " where
        HQL_COUNT_ALL = " select count(*) from " + this.entityClass.getSimpleName();
    }

    @Autowired
    @Qualifier("sessionFactory")
    private SessionFactory sessionFactory;

    public Session getSession() {
        //事务必须是开启的，否则获取不到
        return sessionFactory.getCurrentSession();
    }
……
}
```

Spring3.1集成Hibernate4不再需要HibernateDaoSupport和HibernateTemplate了，直接使用原生API即可。

四、通用**Service**层代码此处省略，看源代码，有了通用代码后**CURD**就不用再写了。

```
@Service("UserService")
public class UserServiceImpl extends BaseService<UserModel, Integer> implements UserServi

    private static final Logger LOGGER = LoggerFactory.getLogger(UserServiceImpl.class);

    private UserDao userDao;

    @Autowired
    @Qualifier("UserDao")
    @Override
    public void setBaseDao(IBaseDao<UserModel, Integer> userDao) {
        this.baseDao = userDao;
        this.userDao = (UserDao) userDao;
    }

    @Override
    public Page<UserModel> query(int pn, int pageSize, UserQueryModel command) {
        return PageUtil.getPage(userDao.countQuery(command) ,pn, userDao.query(pn, pageSi
    }
}
```

五、表现层 **Controller**实现

采用SpringMVC支持的REST风格实现，具体看代码，此处我们使用了java Validator框架 来进行 表现层数据验证

在Model实现上加验证注解

```
    @Pattern(regexp = "[A-Za-z0-9]{5,20}", message = "{username.illegal}") //java validat
    private String username;

    @NotEmpty(message = "{email.illegal}")
    @Email(message = "{email.illegal}") //错误消息会自动到MessageSource中查找
    private String email;

    @Pattern(regexp = "[A-Za-z0-9]{5,20}", message = "{password.illegal}")
    private String password;

    @DateFormat( message="{register.date.error}")//自定义的验证器
    private Date registerDate;
```

在Controller中相应方法的需要验证的参数上加@Valid即可

```
    @RequestMapping(value = "/user/add", method = {RequestMethod.POST})
    public String add(Model model, @ModelAttribute("command") @Valid UserModel command, B
```

六、**Spring**集成测试

使用Spring集成测试能很方便的进行Bean的测试，而且使用@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)能自动回滚事务，清理测试前后状态。

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:spring-config.xml"})
@Transactional
@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)
public class UserServiceTest {

    AtomicInteger counter = new AtomicInteger();

    @Autowired
    private UserService userService;
    ……
}
```

其他部分请直接看源码，欢迎大家讨论。

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/2625.html】

Spring Web MVC中的页面缓存支持 ——跟我学SpringMVC系列

注：本章讲的是**Spring2**的**@Deprecated**，但还是有必要提一下。跟我学**SpringMVC**系列。。

4.2、Controller接口

```
package org.springframework.web.servlet.mvc;
public interface Controller {
       ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse respons
}
```

这是控制器接口，此处只有一个方法handleRequest，用于进行请求的功能处理，处理完请求后返回ModelAndView（Model模型数据部分 和 View视图部分）。

还记得第二章的HelloWorld吗？我们的HelloWorldController实现接口，Spring默认提供了一些Controller接口的实现以方便我们使用，具体继承体系如图4-1：



、WebContentGenerator

用于提供如浏览器缓存控制、是否必须有session开启、支持的请求方法类型（GET、POST等）等，该类主要有如下属性：

Set<String> supportedMethods：设置支持的请求方法类型，默认支持"GET"、"POST"、"HEAD"，如果我们想支持"PUT"，则可以加入该集合"PUT"。

boolean requireSession = false：是否当前请求必须有session，如果此属性为true，但当前请求没有打开session将抛出HttpSessionRequiredException异常；

boolean useExpiresHeader = true：是否使用HTTP1.0协议过期响应头：如果true则会在响应头添加："Expires："；需要配合cacheSeconds使用；

boolean useCacheControlHeader = true：是否使用HTTP1.1协议的缓存控制响应头，如果true则会在响应头添加；需要配合cacheSeconds使用；

boolean useCacheControlNoStore = true：是否使用HTTP 1.1协议的缓存控制响应头，如果true则会在响应头添加；需要配合cacheSeconds使用；

private int cacheSeconds = -1：缓存过期时间，正数表示需要缓存，负数表示不做任何事情（也就是说保留上次的缓存设置），

1、cacheSeconds **=**0时，则将设置如下响应头数据：

Pragma：no-cache // HTTP 1.0的不缓存响应头

Expires：1L // useExpiresHeader=true时，HTTP 1.0

Cache-Control：**no-cache** // useCacheControlHeader=true时，HTTP 1.1

Cache-Control：**no-store** // useCacheControlNoStore=true时，该设置是防止Firefox缓存

2、cacheSeconds>0时，则将设置如下响应头数据：

Expires：System.currentTimeMillis() + **cacheSeconds** * 1000L // useExpiresHeader=true时，HTTP 1.0

Cache-Control：**max-age=cacheSeconds** // useCacheControlHeader=true时，HTTP 1.1

3、cacheSeconds<0时，则什么都不设置，即保留上次的缓存设置。

此处简单说一下以上响应头的作用，缓存控制已超出本书内容：

HTTP1.0缓存控制响应头

Pragma：no-cache：表示防止客户端缓存，需要强制从服务器获取最新的数据；

Expires：HTTP1.0响应头，本地副本缓存过期时间，如果客户端发现缓存文件没有过期则不发送请求，HTTP的日期时间必须是格林威治时间（GMT），如“Expires:Wed, 14 Mar 2012 09:38:32 GMT”；

HTTP1.1缓存控制响应头

Cache-Control：no-cache 强制客户端每次请求获取服务器的最新版本，不经过本地缓存的副本验证；

Cache-Control：no-store强制客户端不保存请求的副本，该设置是防止Firefox缓存

Cache-Control：max-age=[秒] 客户端副本缓存的最长时间，类似于HTTP1.0的Expires，只是此处是基于请求的相对时间间隔来计算，而非绝对时间。

还有相关缓存控制机制如Last-Modified（最后修改时间验证，客户端的上一次请求时间 在 服务器的最后修改时间 之后，说明服务器数据没有发生变化 返回304状态码）、ETag（没有变化时不重新下载数据，返回304）。

该抽象类默认被AbstractController和WebContentInterceptor继承。

4.4、AbstractController

该抽象类实现了Controller，并继承了WebContentGenerator（具有该类的特性，具体请看4.3），该类有如下属性：

boolean synchronizeOnSession = false：表示该控制器是否在执行时同步session，从而保证该会话的用户串行访问该控制器。

```
public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse respons
        //委托给WebContentGenerator进行缓存控制
        checkAndPrepare(request, response, this instanceof LastModified);
        //当前会话是否应串行化访问.
        if (this.synchronizeOnSession) {
            HttpSession session = request.getSession(false);
            if (session != null) {
                Object mutex = WebUtils.getSessionMutex(session);
                synchronized (mutex) {
                    return handleRequestInternal(request, response);
                }
            }
        }
        return handleRequestInternal(request, response);
}
```

可以看出AbstractController实现了一些特殊功能，如继承了WebContentGenerator缓存控制功能，并提供了可选的会话的串行化访问功能。而且提供了handleRequestInternal方法，因此我们应该在具体的控制器类中实现handleRequestInternal方法，而不再是handleRequest。

AbstractController使用方法：

首先让我们使用AbstractController来重写第二章的HelloWorldController：

```
public class HelloWorldController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo
        //1、收集参数
        //2、绑定参数到命令对象
        //3、调用业务对象
        //4、选择下一个页面
        ModelAndView mv = new ModelAndView();
        //添加模型数据 可以是任意的POJO对象
        mv.addObject("message", "Hello World!");
        //设置逻辑视图名，视图解析器会根据该名字解析到具体的视图页面
        mv.setViewName("hello");
        return mv;
    }
}
```

可以看出AbstractController实现了一些特殊功能，如继承了WebContentGenerator缓存控制功能，并提供了可选的会话的串行化访问功能。而且提供了handleRequestInternal方法，因此我们应该在具体的控制器类中实现handleRequestInternal方法，而不再是handleRequest。

AbstractController使用方法：

首先让我们使用AbstractController来重写第二章的HelloWorldController：

```
public class HelloWorldController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo
        //1、收集参数
        //2、绑定参数到命令对象
        //3、调用业务对象
        //4、选择下一个页面
        ModelAndView mv = new ModelAndView();
        //添加模型数据 可以是任意的POJO对象
        mv.addObject("message", "Hello World!");
        //设置逻辑视图名，视图解析器会根据该名字解析到具体的视图页面
        mv.setViewName("hello");
        return mv;
    }
}
```

```
<!— 在chapter3-servlet.xml配置处理器 -->
<bean name="/hello" class="cn.javass.chapter3.web.controller.HelloWorldController"/>
```

从如上代码我们可以看出：

1、继承AbstractController

2、实现handleRequestInternal方法即可。

直接通过response写响应

如果我们想直接在控制器通过response写出响应呢，以下代码帮我们阐述：

```
public class HelloWorldWithoutReturnModelAndViewController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo

        resp.getWriter().write("Hello World!!");
        //如果想直接在该处理器/控制器写响应 可以通过返回null告诉DispatcherServlet自己已经写出响应了，
        return null;
    }
}
```

<!— 在chapter3-servlet.xml配置处理器 -->

```
<bean name="/helloWithoutReturnModelAndView" class="cn.javass.chapter3.web.controller.Hel
```

从如上代码可以看出如果想直接在控制器写出响应，只需要通过response写出，并返回null即可。

强制请求方法类型：

```
<!— 在chapter3-servlet.xml配置处理器 -->
<bean name="/helloWithPOST" class="cn.javass.chapter3.web.controller.HelloWorldController
        <property name="supportedMethods" value="POST"></property>
</bean>
```

以上配置表示只支持POST请求，如果是GET请求客户端将收到"HTTP Status 405 - Request method 'GET' not supported"。

比如注册/登录可能只允许POST请求。

当前请求的**session**前置条件检查，如果当前请求无**session**将抛出**HttpSessionRequiredException**异常：

```
<!— 在chapter3-servlet.xml配置处理器 -->
<bean name="/helloRequireSession"
class="cn.javass.chapter3.web.controller.HelloWorldController">
        <property name="requireSession" value="true"/>
</bean>
```

```
在进入该控制器时，一定要有session存在，否则抛出HttpSessionRequiredException异常。
```

Session同步：

即同一会话只能串行访问该控制器。

缓存控制：

1、 缓存5秒，cacheSeconds=5

```
package cn.javass.chapter3.web.controller;
//省略import
public class HelloWorldCacheController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo

        //点击后再次请求当前页面
        resp.getWriter().write("<a href=''>this</a>");
        return null;
    }
}
```

<!— 在chapter3-servlet.xml配置处理器 -->

```
<bean name="/helloCache"
class="cn.javass.chapter3.web.controller.HelloWorldCacheController">
<property name="cacheSeconds" value="5"/>
</bean>
```

如上配置表示告诉浏览器缓存**5**秒钟：

开启chrome浏览器调试工具：



服务器返回的响应头如下所示：



添加了“Expires:Wed, 14 Mar 2012 09:38:32 GMT” 和“Cache-Control:max-age=5” 表示允许客户端缓存5秒，当你点“this”链接时，会发现如下：



而且服务器也没有收到请求，当过了5秒后，你再点“this”链接会发现又重新请求服务器下载新数据。

注：下面提到一些关于缓存控制的一些特殊情况：

1、对于一般的页面跳转（如超链接点击跳转、通过js调用window.open打开新页面都是会使用浏览器缓存的，在未过期情况下会直接使用浏览器缓存的副本，在未过期情况下一次请求也不发送）；

2、对于刷新页面（如按F5键刷新），会再次发送一次请求到服务器的；

2、不缓存，**cacheSeconds=0**

<!— 在chapter3-servlet.xml配置处理器 -->

```
<bean name="/helloNoCache"
class="cn.javass.chapter3.web.controller.HelloWorldCacheController">
<property name="cacheSeconds" value="0"/>
</bean>
```

以上配置会要求浏览器每次都去请求服务器下载最新的数据：



3、**cacheSeconds<0**，将不添加任何数据

响应头什么缓存控制信息也不加。

4、**Last-Modified**缓存机制

（1、在客户端第一次输入url时，服务器端会返回内容和状态码200表示请求成功并返回了内容；同时会添加一个“Last-Modified”的响应头表示此文件在服务器上的最后更新时间，如“Last-Modified:Wed, 14 Mar 2012 10:22:42 GMT”表示最后更新时间为（2012-03-14 10：22）；

（2、客户端第二次请求此URL时，客户端会向服务器发送请求头 “If-Modified-Since”，询问服务器该时间之后当前请求内容是否有被修改过，如“If-Modified-Since: Wed, 14 Mar 2012 10:22:42 GMT”，如果服务器端的内容没有变化，则自动返回 HTTP 304状态码（只要响应

头，内容为空，这样就节省了网络带宽）。

客户端强制缓存过期：

（1、可以按ctrl+F5强制刷新（会添加请求头 HTTP1.0 Pragma:no-cache和 HTTP1.1 Cache-Control:no-cache 、If-Modified-Since请求头被删除）表示强制获取服务器内容，不缓存。

（2、在请求的url后边加上时间戳来重新获取内容，加上时间戳后浏览器就认为不是同一份内容：

http://sishuok.com/?2343243243 和 http://sishuok.com/?34334344 是两次不同的请求。

Spring也提供了**Last-Modified**机制的支持，只需要实现**LastModified**接口，如下所示：

package cn.javass.chapter3.web.controller;

```
public class HelloWorldLastModifiedCacheController extends AbstractController implements
    private long lastModified;
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo
        //点击后再次请求当前页面
        resp.getWriter().write("<a href=''>this</a>");
        return null;
    }
    public long getLastModified(HttpServletRequest request) {
        if(lastModified == 0L) {
            //TODO 此处更新的条件：如果内容有更新，应该重新返回内容最新修改的时间戳
            lastModified = System.currentTimeMillis();
        }
        return lastModified;
    }
}
```

<!— 在chapter3-servlet.xml配置处理器 -->

```
<bean name="/helloLastModified"
class="cn.javass.chapter3.web.controller.HelloWorldLastModifiedCacheController"/>
```

HelloWorldLastModifiedCacheController只需要实现LastModified接口的getLastModified方法，保证当内容发生改变时返回最新的修改时间即可。

分析：

（1、发送请求到服务器，如（http://localhost:9080/springmvc-chapter3/helloLastModified），则服务器返回的响应为：



（2、再次按F5刷新客户端，返回状态码304表示服务器没有更新过：



（3、重启服务器，再次刷新，会看到200状态码（因为服务器的lastModified时间变了）。

Spring判断是否过期，通过如下代码，即请求的**“If-Modified-Since”** 大于等于当前的**getLastModified**方法的时间戳，则认为没有修改：

this.notModified = (ifModifiedSince >= (lastModifiedTimestamp / 1000 * 1000));

5、ETag（实体标记）缓存机制

（1：浏览器第一次请求，服务器在响应时给请求URL标记，并在HTTP响应头中将其传送到客户端，类似服务器端返回的格式：“ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"”

（2：浏览器第二次请求，客户端的查询更新格式是这样的：“If-None-Match:"0f8b0c86fe2c0c7a67791e53d660208e3"”，如果ETag没改变，表示内容没有发生改变，则返回状态304。

Spring也提供了对**ETag**的支持，具体需要在**web.xml**中配置如下代码：

```
<filter>
   <filter-name>etagFilter</filter-name>
   <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>etagFilter</filter-name>
   <servlet-name>chapter3</servlet-name>
</filter-mapping>
```

此过滤器只过滤到我们DispatcherServlet的请求。

分析：

1）：发送请求到服务器：“http://localhost:9080/springmvc-chapter3/hello”，服务器返回的响应头中添加了（ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"）：



2）：浏览器再次发送请求到服务器（按F5刷新），请求头中添加了“If-None-Match:

"0f8b0c86fe2c0c7a67791e53d660208e3"”，响应返回304代码，表示服务器没有修改，并且响应头再次添加了“ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"”（每次都需要计算）：



那服务器端是如何计算**ETag**的呢？

```
protected String generateETagHeaderValue(byte[] bytes) {
        StringBuilder builder = new StringBuilder("\"0");
        DigestUtils.appendMd5DigestAsHex(bytes, builder);
        builder.append('"');
        return builder.toString();
}
```

bytes是response要写回到客户端的响应体（即响应的内容数据），是通过MD5算法计算的内容的摘要信息。也就是说如果服务器内容不发生改变，则ETag每次都是一样的，即服务器端的内容没有发生改变。

此处只列举了部分缓存控制，详细介绍超出了本书的范围，强烈推荐：**http://www.mnot.net/cache_docs/**（中文版**http://www.chedong.com/tech/cache_docs.html**）详细了解**HTTP**缓存控制及为什么要缓存。

缓存的目的是减少相应延迟 和 减少网络带宽消耗，比如**css**、**js**、图片这类静态资源应该进行缓存。

实际项目一般使用反向代理服务器（如**nginx**、**apache**等）进行缓存。

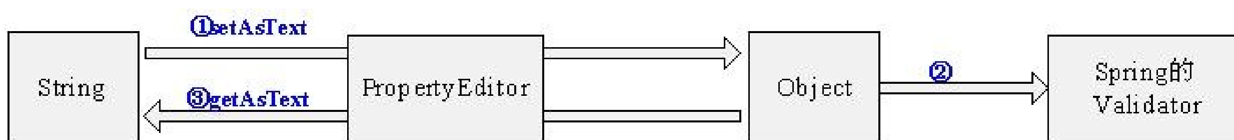
Spring3 Web MVC下的数据类型转换（第一篇）——《跟我学Spring3 Web MVC》抢先看

基于spring-framework-3.1.1.RELEASE

7.1、简介

在编写可视化界面项目时，我们通常需要对数据进行类型转换、验证及格式化。

一、在**Spring3**之前，我们使用如下架构进行类型转换、验证及格式化：



流程：

①：类型转换：首先调用PropertyEditor的setAsText（String），内部根据需要调用setValue(Object)方法进行设置转换后的值；

②：数据验证：需要显示调用Spring的Validator接口实现进行数据验证；

③：格式化显示：需要调用PropertyEditor的getText进行格式化显示。

使用如上架构的缺点是：

（1、PropertyEditor被设计为只能String<——>Object之间转换，不能任意对象类型<——>任意类型，如我们常见的Long时间戳到Date类型的转换是办不到的；

（2、PropertyEditor是线程不安全的，也就是有状态的，因此每次使用时都需要创建一个，不可重用；

（3、PropertyEditor不是强类型的，setValue（Object）可以接受任意类型，因此需要我们自己判断类型是否兼容；

（4、需要自己编程实现验证，Spring3支持更棒的注解验证支持；

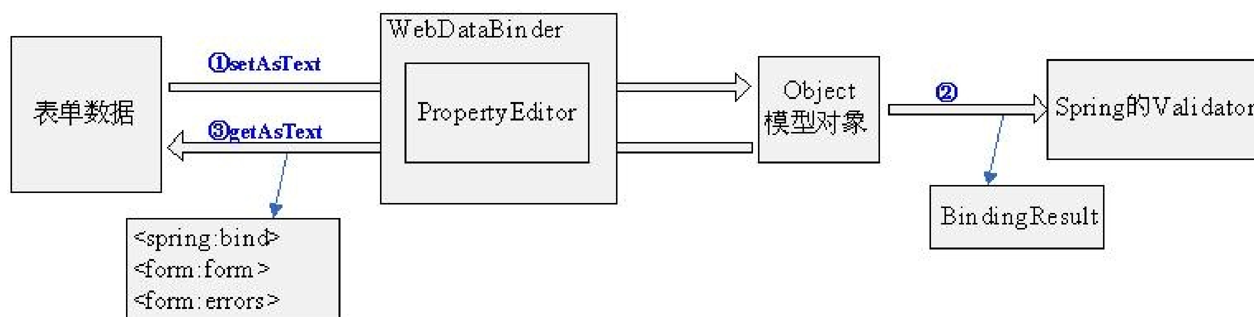
（5、在使用SpEL表达式语言或DataBinder时，只能进行String<--->Object之间的类型转换；

```
（6
```

、

```
不支持细粒度的类型转换/格式化，如UserModel的registerDate需要转换/格式化类似“``2012-05-01``”的
```

```
**在Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：**
```



流程：

①、类型转换：首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过PropertyEditor实

②：数据验证：在控制器中的功能处理方法中，需要显示的调用Spring的Validator实现并将错误信息添加到BindingResult对象中；

③：格式化显示：在表单页面可以通过如下方式展示通过 `PropertyEditor` 格式化的数据和错误信息：

```
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

首先需要通过如上taglib指令引入spring的两个标签库。

```
//1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）
<spring:bind path="dataBinderTest.phoneNumber">${status.value}</spring:bind>
```

```
//2、<spring:eval>标签，自动调用ConversionService并选择相应的Converter SPI进行格式化展示
<spring:eval expression="dataBinderTest.phoneNumber"></spring:eval>
```

如上代码能工作的前提是在RequestMappingHandlerMapping配置了ConversionServiceExposingInterceptor，它的作用是暴露conversionService到请求中以便如<spring:eval>标签使用。

```
//3、通过form标签，内部的表单标签会自动调用命令/表单对象属性对应的PropertyEditor进行格式化显示
<form:form commandName="dataBinderTest">
    <form:input path="phoneNumber"/><!-- 如果出错会显示错误之前的数据而不是空 -->
</form:form>
```

```
//4、显示验证失败后的错误信息
<form:errors></form:errors>
```

接下来我们就详细学习一下这些知识吧。

7.2、数据类型转换

7.2.1、Spring3之前的PropertyEditor

PropertyEditor介绍请参考【4.16.1、数据类型转换】。

一、测试之前我们需要准备好测试环境：

（1、模型对象，和【4.16.1、数据类型转换】使用的一样，需要将DataBinderTestModel模型类及相关类拷贝过来放入cn.javass.chapter7.model包中。

（2、控制器定义：

```
package cn.javass.chapter7.web.controller;
//省略import
@Controller
public class DataBinderTestController {
    @RequestMapping(value = "/dataBind")
    public String test(DataBinderTestModel command) {
            //输出command对象看看是否绑定正确
        System.out.println(command);
        model.addAttribute("dataBinderTest", command);
        return "bind/success";
    }
}
```

(3、Spring配置文件定义，请参考chapter7-servlet.xml，并注册DataBinderTestController：

```
<bean class="cn.javass.chapter7.web.controller.DataBinderTestController"/>
```

（4、测试的URL：

http://localhost:9080/springmvc-chapter7/dataBind?username=zhang&bool=yes&schoolInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&phoneNumber=010-12345678&date=2012-3-18 16:48:48&state=blocked

二、注解式控制器注册**PropertyEditor**：

1、使用**WebDataBinder**进行控制器级别注册**PropertyEditor**（控制器独享）

```
@InitBinder
//此处的参数也可以是ServletRequestDataBinder类型
public void initBinder(WebDataBinder binder) throws Exception {
    //注册自定义的属性编辑器
    //1、日期
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    CustomDateEditor dateEditor = new CustomDateEditor(df, true);
    //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
    binder.registerCustomEditor(Date.class, dateEditor);
    //自定义的电话号码编辑器(和【4.16.1、数据类型转换】一样)
    binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());
}
```

和【4.16.1、数据类型转换】一节类似，只是此处需要通过@InitBinder来注册自定义的PropertyEditor。

2、使用 `**WebBindingInitializer批量注册**` **PropertyEditor**

和【4.16.1、数据类型转换】不太一样，因为我们的注解式控制器是POJO，没有实现任何东西，因此无法注入WebBindingInitializer，此时我们需要把WebBindingInitializer注入到我们的RequestMappingHandlerAdapter或AnnotationMethodHandlerAdapter，这样对于所有的注解式控制器都是共享的。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerA
    <property name="webBindingInitializer">
        <bean class="cn.javass.chapter7.web.controller.support.initializer.MyWebBindingIn
    </property>
</bean>
```

此时我们注释掉控制器级别通过@InitBinder注册PropertyEditor的方法。

3、全局级别注册PropertyEditor（全局共享）

和【4.16.1、数据类型转换】一节一样，此处不再重复。请参考【4.16.1、数据类型转换】的【全局级别注册PropertyEditor（全局共享）】。

`接下来我们看一下Spring3提供的更强大的类型转换支持。`

7.2.2、Spring3开始的类型转换系统

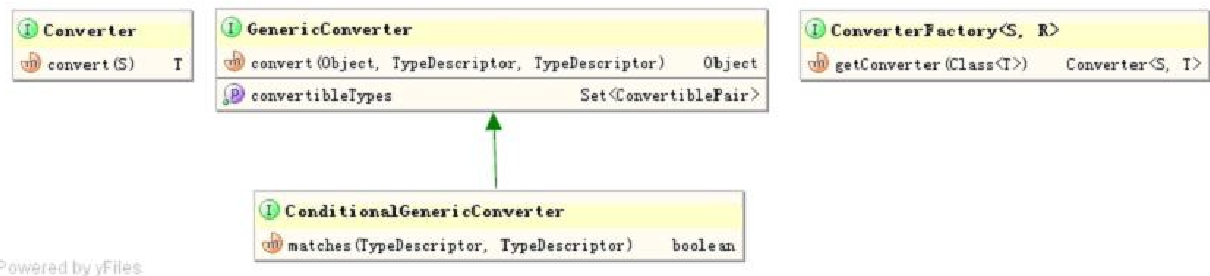
Spring3引入了更加通用的类型转换系统，其定义了SPI接口（Converter等）和相应的运行时执行类型转换的API（ConversionService等），在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来转换外部Bean属性的值到Bean属性需要的类型。

该类型转换系统是Spring通用的，其定义在org.springframework.core.convert包中，不仅仅在Spring Web MVC场景下。目标是完全替换PropertyEditor，提供无状态、强类型且可以在任意类型之间转换的类型转换系统，可以用于任何需要的地方，如SpEL、数据绑定。

Converter SPI完成通用的类型转换逻辑，如java.util.Date<---->java.lang.Long或java.lang.String---->PhoneNumberModel等。

7.2.2.1、架构

1、类型转换器：提供类型转换的实现支持。



一个有如下三种接口：

（**1、Converter**：类型转换器，用于转换S类型到T类型，此接口的实现必须是线程安全的且可以被共享。

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> { //① S是源类型 T是目标类型
    T convert(S source); //② 转换S类型的source到T目标类型的转换方法
}
```

示例：请参考cn.javass.chapter7.converter.support.StringToPhoneNumberConverter转换器，用于将String--->PhoneNumberModel。

此处我们可以看到Converter接口实现只能转换一种类型到另一种类型，不能进行多类型转换，如将一个数组转换成集合，如（String[] ----> List<String>、String[]----->List<PhoneNumberModel>等）。

（**2、GenericConverter和ConditionalGenericConverter**：GenericConverter接口实现能在多种类型之间进行转换，ConditionalGenericConverter是有条件的在多种类型之间进行转换。

```
package org.springframework.core.convert.converter;
public interface GenericConverter {
    Set<ConvertiblePair> getConvertibleTypes();
    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

getConvertibleTypes:指定了可以转换的目标类型对；

convert：在sourceType和targetType类型之间进行转换。

```
package org.springframework.core.convert.converter;
public interface ConditionalGenericConverter extends GenericConverter {
    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

matches：用于判断sourceType和targetType类型之间能否进行类型转换。

示例：如org.springframework.core.convert.support.ArrayToCollectionConverter和CollectionToArrayConverter用于在数组和集合间进行转换的ConditionalGenericConverter实现，如在String[]<---->List<String>、String[]<---->List<PhoneNumberModel>等之间进行类型

转换。

对于我们大部分用户来说一般不需要自定义GenericConverter, 如果需要可以参考内置的GenericConverter来实现自己的。

（**3**、**ConverterFactory**：工厂模式的实现，用于选择将一种S源类型转换为R类型的子类型T的转换器的工厂接口。

```
package org.springframework.core.convert.converter;
public interface ConverterFactory<S, R> {
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

S：源类型；R目标类型的父类型；T：目标类型，且是R类型的子类型；

getConverter：得到目标类型的对应的转换器。

示例：如org.springframework.core.convert.support.NumberToNumberConverterFactory用于在Number类型子类型之间进行转换，如Integer--->Double， Byte---->Integer， Float--->Double等。

对于我们大部分用户来说一般不需要自定义ConverterFactory，如果需要可以参考内置的ConverterFactory来实现自己的。

2、类型转换器注册器、类型转换服务：提供类型转换器注册支持，运行时类型转换API支持。



一共有如下两种接口：

（**1**、**ConverterRegistry**：类型转换器注册支持，可以注册/删除相应的类型转换器。

```
package org.springframework.core.convert.converter;
public interface ConverterRegistry {
    void addConverter(Converter<?, ?> converter);
    void addConverter(Class<?> sourceType, Class<?> targetType, Converter<?, ?> converter
    void addConverter(GenericConverter converter);
    void addConverterFactory(ConverterFactory<?, ?> converterFactory);
    void removeConvertible(Class<?> sourceType, Class<?> targetType);
}
```

可以注册：Converter实现，GenericConverter实现，ConverterFactory实现。

（**2**、**ConversionService**：运行时类型转换服务接口，提供运行期类型转换的支持。

```
package org.springframework.core.convert;
public interface ConversionService {
    boolean canConvert(Class<?> sourceType, Class<?> targetType);
    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);
    <T> T convert(Object source, Class<T> targetType);
    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

convert：将源对象转换为目标类型的目标对象。

Spring提供了两个默认实现（其都实现了ConverterRegistry、ConversionService接口）：

DefaultConversionService:默认的类型转换服务实现；

DefaultFormattingConversionService：带数据格式化支持的类型转换服务实现，一般使用该服务实现即可。

7.2.2.2、Spring内建的类型转换器如下所示：

类名	说明
第一组：标量转换器	
StringToBooleanConverter	String----->Booleantrue:true/on/yes/1；false:false/off/no/0
ObjectToStringConverter	Object----->String调用toString方法转换
StringToNumberConverterFactory	String----->Number（如Integer、Long等）
NumberToNumberConverterFactory	Number子类型(Integer、Long、Double等)<——> Number子类型(Integer、Long、Double等)
StringToCharacterConverter	String----->java.lang.Character取字符串第一个字符
NumberToCharacterConverter	Number子类型(Integer、Long、Double等)——>java.lang.Character
CharacterToNumberFactory	java.lang.Character ——>Number子类型(Integer、Long、Double等)

StringToEnumConverterFactory	String----->enum类型通过Enum.valueOf将字符串转换为需要的enum类型
EnumToStringConverter	enum类型----->String返回enum对象的name()值
StringToLocaleConverter	String----->java.util.Local
PropertiesToStringConverter	java.util.Properties----->String默认通过ISO-8859-1解码
StringToPropertiesConverter	String----->java.util.Properties默认使用ISO-8859-1编码
第二组：集合、数组相关转换器	
ArrayToCollectionConverter	任意S数组---->任意T集合（List、Set）
CollectionToArrayConverter	任意T集合（List、Set）---->任意S数组
ArrayToArrayConverter	任意S数组<---->任意T数组
CollectionToCollectionConverter	任意T集合（List、Set）<---->任意T集合（List、Set）即集合之间的类型转换
MapToMapConverter	Map<---->Map之间的转换
ArrayToStringConverter	任意S数组---->String类型
StringToArrayConverter	String----->数组默认通过“,”分割，且去除字符串的两边空格(trim)
ArrayToObjectConverter	任意S数组---->任意Object的转换(如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换)
ObjectToArrayConverter	Object----->单元素数组
CollectionToStringConverter	任意T集合（List、Set）---->String类型
StringToCollectionConverter	String----->集合（List、Set）默认通过“,”分割，且去除字符串的两边空格(trim)
CollectionToObjectConverter	任意T集合---->任意Object的转换(如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换)
ObjectToCollectionConverter	Object----->单元素集合
第三组：默认（**fallback**）转换器：之前的转换器不能转换时调用	
ObjectToObjectConverter	Object（S）----->Object（T）首先尝试valueOf进行转换、没有则尝试new 构造器(S)
IdToEntityConverter	Id(S)----->Entity(T)查找并调用public static T findEntityName获取目标对象，EntityName是T类型的简单类型
FallbackObjectToStringConverter	Object----->StringConversionService作为恢复使用，即其他转换器不能转换时调用（执行对象的

	toString()方法）

S：代表源类型，T：代表目标类型

如上的转换器在使用转换服务实现DefaultConversionService和DefaultFormattingConversionService时会自动注册。

7.2.2.3、示例

（1、自定义String----->PhoneNumberModel的转换器

```
package cn.javass.chapter7.web.controller.support.converter;
//省略import
public class StringToPhoneNumberConverter implements Converter<String, PhoneNumberModel>
    Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
    @Override
    public PhoneNumberModel convert(String source) {
        if(!StringUtils.hasLength(source)) {
            //①如果source为空 返回null
            return null;
        }
        Matcher matcher = pattern.matcher(source);
        if(matcher.matches()) {
            //②如果匹配 进行转换
            PhoneNumberModel phoneNumber = new PhoneNumberModel();
            phoneNumber.setAreaCode(matcher.group(1));
            phoneNumber.setPhoneNumber(matcher.group(2));
            return phoneNumber;
        } else {
            //③如果不匹配 转换失败
            throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-1234
        }
    }
}
```

String转换为Date的类型转换器，请参考cn.javass.chapter7.web.controller.support.converter.StringToDateConverter。

(2、测试用例(cn.javass.chapter7.web.controller.support.converter.ConverterTest)

```
@Test
public void testStringToPhoneNumberConvert() {
    DefaultConversionService conversionService = new DefaultConversionService();
    conversionService.addConverter(new StringToPhoneNumberConverter());

    String phoneNumberStr = "010-12345678";
    PhoneNumberModel phoneNumber = conversionService.convert(phoneNumberStr, PhoneNumberM

    Assert.assertEquals("010", phoneNumber.getAreaCode());
}
```

类似于PhoneNumberEditor将字符串“010-12345678”转换为PhoneNumberModel。

```
@Test
public void testOtherConvert() {
    DefaultConversionService conversionService = new DefaultConversionService();

    //"1"--->true（字符串“1”可以转换为布尔值true）
    Assert.assertEquals(Boolean.valueOf(true), conversionService.convert("1", Boolean.cla

    //"1,2,3,4"--->List（转换完毕的集合大小为4）
    Assert.assertEquals(4, conversionService.convert("1,2,3,4", List.class).size());
}
```

其他类型转换器使用也是类似的，此处不再重复。

7.2.2.4、集成到Spring Web MVC环境

（1、注册ConversionService实现和自定义的类型转换器

```
<!-- ①注册ConversionService -->
<bean id="conversionService" class="org.springframework.format.support.
                                                       FormattingConversionServiceFactory
    <property name="converters">
       <list>
            <bean class="cn.javass.chapter7.web.controller.support.
                                   converter.StringToPhoneNumberConverter"/>
            <bean class="cn.javass.chapter7.web.controller.support.
                                   converter.StringToDateConverter">
                <constructor-arg value="yyyy-MM-dd"/>
            </bean>
        </list>
    </property>
</bean>
```

FormattingConversionServiceFactoryBean：是FactoryBean实现，默认使用DefaultFormattingConversionService转换器服务实现；

converters：注册我们自定义的类型转换器，此处注册了String--->PhoneNumberModel和String--->Date的类型转换器。

（2、通过ConfigurableWebBindingInitializer注册ConversionService

```
<!-- ②使用ConfigurableWebBindingInitializer注册conversionService -->
<bean id="webBindingInitializer" class="org.springframework.web.bind.support.
                                                                        ConfigurableWebBi
    <property name="conversionService" ref="conversionService"/>
</bean>
```

此处我们通过ConfigurableWebBindingInitializer绑定初始化器进行ConversionService的注册；

3、注册ConfigurableWebBindingInitializer到RequestMappingHandlerAdapter

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.
                                                           RequestMappingHandlerAdapter"
<property name="webBindingInitializer" ref="webBindingInitializer"/>
</bean>
```

通过如上配置，我们就完成了Spring3.0的类型转换系统与Spring Web MVC的集成。此时可以启动服务器输入之前的URL测试了。

此时可能有人会问，如果我同时使用PropertyEditor和ConversionService，执行顺序是什么呢？内部首先查找PropertyEditor进行类型转换，如果没有找到相应的PropertyEditor再通过ConversionService进行转换。

如上集成过程看起来比较麻烦，后边我们会介绍<mvc:annotation-driven>和@EnableWebMvc，ConversionService会自动注册，后续章节再详细介绍。

Spring3 Web MVC下的数据格式化（第二篇）——《跟我学Spring3 Web MVC》抢先看

基于spring-framework-3.1.1.RELEASE

7.3、数据格式化

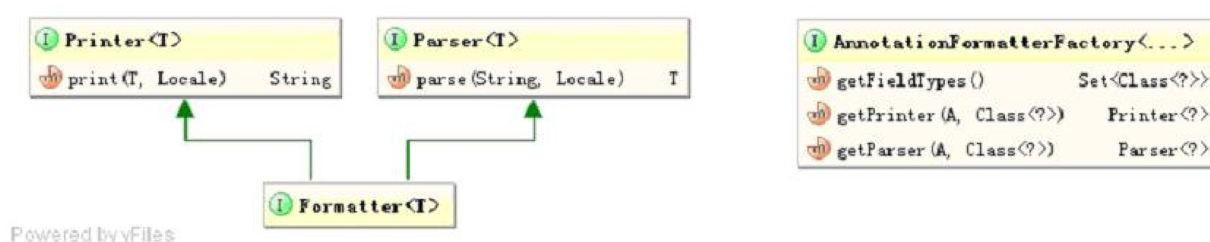
在如Web /客户端项目中，通常需要将数据转换为具有某种格式的字符串进行展示，因此上节我们学习的数据类型转换系统核心作用不是完成这个需求，因此Spring3引入了格式化转换器（Formatter SPI） 和格式化服务API（FormattingConversionService）从而支持这种需求。在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来进行对象的解析和格式化，而且支持细粒度的字段级别的格式化/解析。

Formatter SPI核心是完成解析和格式化转换逻辑，在如Web应用/客户端项目中，需要解析、打印/展示本地化的对象值时使用，如根据Locale信息将java.util.Date---->java.lang.String打印/展示、java.lang.String---->java.util.Date等。

该格式化转换系统是Spring通用的，其定义在org.springframework.format包中，不仅仅在Spring Web MVC场景下。

7.3.1、架构

1、格式化转换器：提供格式化转换的实现支持。



一共有如下两组四个接口：

（1、**Printer**接口：格式化显示接口，将T类型的对象根据Locale信息以某种格式进行打印显示（即返回字符串形式）；

```
package org.springframework.format;
public interface Printer<T> {
    String print(T object, Locale locale);
}
```

（2、**Parser**接口：解析接口，根据Locale信息解析字符串到T类型的对象；

```
package org.springframework.format;
public interface Parser<T> {
    T parse(String text, Locale locale) throws ParseException;
}
```

解析失败可以抛出java.text.ParseException或IllegalArgumentException异常即可。

（3、**Formatter**接口：格式化SPI接口，继承Printer和Parser接口，完成T类型对象的格式化和解析功能；

```
package org.springframework.format;
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

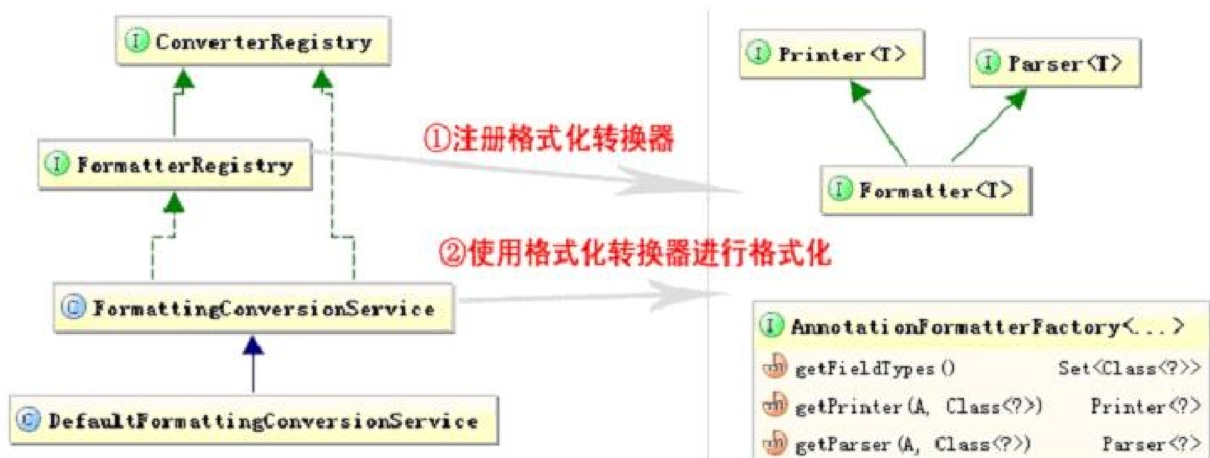
（4、**AnnotationFormatterFactory**接口：注解驱动的字段格式化工厂，用于创建带注解的对象字段的Printer和Parser，即用于格式化和解析带注解的对象字段。

```
package org.springframework.format;
public interface AnnotationFormatterFactory<A extends Annotation> {//①可以识别的注解类型
    Set<Class<?>> getFieldTypes();//②可以被A注解类型注解的字段类型集合
    Printer<?> getPrinter(A annotation, Class<?> fieldType);//③根据A注解类型和fieldType类型
    Parser<?> getParser(A annotation, Class<?> fieldType);//④根据A注解类型和fieldType类型获取

}
```

返回用于格式化和解析被A注解类型注解的字段值的Printer和Parser。如JodaDateTimeFormatAnnotationFormatterFactory可以为带有@DateTimeFormat注解的java.util.Date字段类型创建相应的Printer和Parser进行格式化和解析。

2、格式化转换器注册器、格式化服务：提供类型转换器注册支持，运行时类型转换API支持。



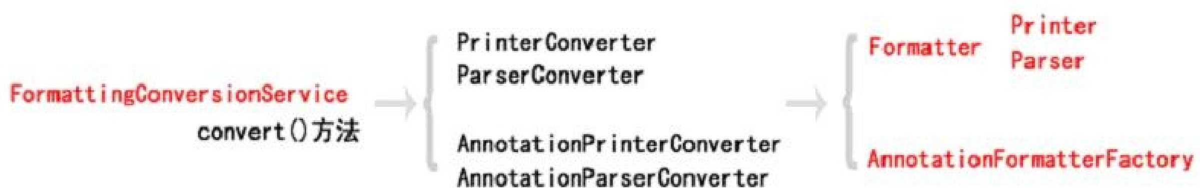
一个有如下两种接口：

（1、FormatterRegistry：格式化转换器注册器，用于注册格式化转换器（Formatter、Printer和Parser、AnnotationFormatterFactory）；

```
package org.springframework.format;
public interface FormatterRegistry extends ConverterRegistry {
    //①添加格式化转换器（Spring3.1 新增API）
    void addFormatter(Formatter<?> formatter);
    //②为指定的字段类型添加格式化转换器
    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);
    //③为指定的字段类型添加Printer和Parser
    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parse
    //④添加注解驱动的字段格式化工厂AnnotationFormatterFactory
    void addFormatterForFieldAnnotation(
                AnnotationFormatterFactory<? extends Annotation> annotationFormatterFacto
}
```

（**2、FormattingConversionService**：继承自ConversionService，运行时类型转换和格式化服务接口，提供运行期类型转换和格式化的支持。

FormattingConversionService内部实现如下图所示：



我们可以看到FormattingConversionService内部实现如上所示，当你调用convert方法时：

(1)若是S类型----->String：调用私有的静态内部类PrinterConverter，其又调用相应的Printer的实现进行格式化；

(2)若是String----->T类型：调用私有的静态内部类ParserConverter，其又调用相应的Parser的实现进行解析；

(3)若是A注解类型注解的S类型----->String：调用私有的静态内部类AnnotationPrinterConverter，其又调用相应的AnnotationFormatterFactory的getPrinter获取Printer的实现进行格式化；

(4)若是String----->A注解类型注解的T类型：调用私有的静态内部类AnnotationParserConverter，其又调用相应的AnnotationFormatterFactory的getParser获取Parser的实现进行解析。

注：S类型表示源类型，T类型表示目标类型，A表示注解类型。

此处可以可以看出之前的Converter SPI完成任意Object与Object之间的类型转换，而Formatter SPI完成任意Object与String之间的类型转换（即格式化和解析，与PropertyEditor类似）。

7.3.2、Spring内建的格式化转换器如下所示：

类名	说明
DateFormatter	java.util.Date<---->String实现日期的格式化/解析
NumberFormatter	java.lang.Number<---->String实现通用样式的格式化/解析
CurrencyFormatter	java.lang.BigDecimal<---->String实现货币样式的格式化/解析
PercentFormatter	java.lang.Number<---->String实现百分数样式的格式化/解析
NumberFormatAnnotationFormatterFactory	@NumberFormat注解类型的数字字段类型<---->String①通过@NumberFormat指定格式化/解析格式②可以格式化/解析的数字类型：Short、Integer、Long、Float、Double、BigDecimal、BigInteger
JodaDateTimeFormatAnnotationFormatterFactory	@DateTimeFormat注解类型的日期字段类型<---->String①通过@DateTimeFormat指定格式化/解析格式②可以格式化/解析的日期类型：joda中的日期类型（org.joda.time包中的）：LocalDate、LocalDateTime、LocalTime、ReadableInstantjava内置的日期类型：Date、Calendar、Longclasspath中必须有Joda-Time类库，否则无法格式化日期类型

NumberFormatAnnotationFormatterFactory和JodaDateTimeFormatAnnotationFormatterFactory（如果classpath提供了Joda-Time类库）在使用格式化服务实现DefaultFormattingConversionService时会自动注册。

7.3.3、示例

在示例之前，我们需要到http://joda-time.sourceforge.net/下载Joda-Time类库，本书使用的是joda-time-2.1版本，将如下jar包添加到classpath：

joda-time-2.1.jar

7.3.3.1、类型级别的解析/格式化

一、直接使用**Formatter SPI**进行解析/格式化

```
//二、CurrencyFormatter：实现货币样式的格式化/解析
CurrencyFormatter currencyFormatter = new CurrencyFormatter();
currencyFormatter.setFractionDigits(2);//保留小数点后几位
currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）

//1、将带货币符号的字符串“$123.125”转换为BigDecimal("123.00")
Assert.assertEquals(new BigDecimal("123.13"), currencyFormatter.parse("$123.125", Locale.
//2、将BigDecimal("123")格式化为字符串“$123.00”展示
Assert.assertEquals("$123.00", currencyFormatter.print(new BigDecimal("123"), Locale.US))
Assert.assertEquals("￥123.00", currencyFormatter.print(new BigDecimal("123"), Locale.CHI
Assert.assertEquals("￥123.00", currencyFormatter.print(new BigDecimal("123"), Locale.JAP
```

parse方法：将带格式的字符串根据Locale信息解析为相应的BigDecimal类型数据；

print方法：将BigDecimal类型数据根据Locale信息格式化为字符串数据进行展示。

不同于Convert SPI，Formatter SPI可以根据本地化（Locale）信息进行解析/格式化。

其他测试用例请参考cn.javass.chapter7.web.controller.support.formatter.InnerFormatterTest的testNumber测试方法和testDate测试方法。

二、使用**DefaultFormattingConversionService**进行解析/格式化

```
@Test
public void testWithDefaultFormattingConversionService() {
    DefaultFormattingConversionService conversionService = new DefaultFormattingConversio
    //默认不自动注册任何Formatter
    CurrencyFormatter currencyFormatter = new CurrencyFormatter();
    currencyFormatter.setFractionDigits(2);//保留小数点后几位
    currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）
    //注册Formatter SPI实现
    conversionService.addFormatter(currencyFormatter);

    //绑定Locale信息到ThreadLocal
    //FormattingConversionService内部自动获取作为Locale信息，如果不设值默认是 Locale.getDefault
    LocaleContextHolder.setLocale(Locale.US);
    Assert.assertEquals("$1,234.13", conversionService.convert(new BigDecimal("1234.128")
    LocaleContextHolder.setLocale(null);

        LocaleContextHolder.setLocale(Locale.CHINA);
        Assert.assertEquals("￥1,234.13", conversionService.convert(new BigDecimal("1234.1
    Assert.assertEquals(new BigDecimal("1234.13"), conversionService.convert("￥1,234.13"
    LocaleContextHolder.setLocale(null);}
```

DefaultFormattingConversionService：带数据格式化功能的类型转换服务实现；

conversionService.addFormatter()：注册Formatter SPI实现；

conversionService.convert(**new** BigDecimal("1234.128"), String.**class**)：用于将BigDecimal类型数据格式化为字符串类型，此处根据“LocaleContextHolder.*setLocale*(locale)”设置的本地化信息进行格式化；

conversionService.convert("￥1,234.13", BigDecimal.**class**)：用于将字符串类型数据解析为BigDecimal类型数据，此处也是根据“LocaleContextHolder.*setLocale*(locale)”设置的本地化信息进行解；

LocaleContextHolder.*setLocale*(locale)：设置本地化信息到ThreadLocal，以便Formatter SPI根据本地化信息进行解析/格式化；

具体测试代码请参考cn.javass.chapter7.web.controller.support.formatter.InnerFormatterTest的testWithDefaultFormattingConversionService测试方法。

三、自定义**Formatter**进行解析/格式化

此处以解析/格式化PhoneNumberModel为例。

（**1**、定义**Formatter SPI**实现

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
public class PhoneNumberFormatter implements Formatter<PhoneNumberModel> {
    Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
    @Override
    public String print(PhoneNumberModel phoneNumber, Locale locale) {//①格式化
        if(phoneNumber == null) {
            return "";
        }
        return new StringBuilder().append(phoneNumber.getAreaCode()).append("-")
                                  .append(phoneNumber.getPhoneNumber()).toString();
    }

    @Override
    public PhoneNumberModel parse(String text, Locale locale) throws ParseException {//②
        if(!StringUtils.hasLength(text)) {
            //①如果source为空 返回null
            return null;
        }
        Matcher matcher = pattern.matcher(text);
        if(matcher.matches()) {
            //②如果匹配 进行转换
            PhoneNumberModel phoneNumber = new PhoneNumberModel();
            phoneNumber.setAreaCode(matcher.group(1));
            phoneNumber.setPhoneNumber(matcher.group(2));
            return phoneNumber;
        } else {
            //③如果不匹配 转换失败
            throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-1234
        }
    }
}
```

类似于Convert SPI实现，只是此处的相应方法会传入Locale本地化信息，这样可以为不同地区进行解析/格式化数据。

（**2**、测试用例：

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
public class CustomerFormatterTest {
    @Test
    public void test() {
        DefaultFormattingConversionService conversionService = new DefaultFormattingConve
        conversionService.addFormatter(new PhoneNumberFormatter());

        PhoneNumberModel phoneNumber = new PhoneNumberModel("010", "12345678");
        Assert.assertEquals("010-12345678", conversionService.convert(phoneNumber, String

        Assert.assertEquals("010", conversionService.convert("010-12345678", PhoneNumberM
    }
}
```

通过PhoneNumberFormatter可以解析String--->PhoneNumberModel和格式化PhoneNumberModel--->String。

到此，类型级别的解析/格式化我们就介绍完了，从测试用例可以看出类型级别的是对项目中的整个类型实施相同的解析/格式化逻辑。

有的同学可能需要在不同的类的字段实施不同的解析/格式化逻辑，如用户模型类的注册日期字段只需要如“2012-05-02”格式进行解析/格式化即可，而订单模型类的下订单日期字段可能需要如“2012-05-02 20：13：13”格式进行展示。

接下来我们学习一下如何进行字段级别的解析/格式化吧。

7.3.3.2、字段级别的解析/格式化

一、使用内置的注解进行字段级别的解析/格式化：

（1、测试模型类准备：

```
package cn.javass.chapter7.model;
public class FormatterModel {
    @NumberFormat(style=Style.NUMBER, pattern="#,###")
    private int totalCount;
    @NumberFormat(style=Style.PERCENT)
    private double discount;
    @NumberFormat(style=Style.CURRENCY)
    private double sumMoney;

    @DateTimeFormat(iso=ISO.DATE)
    private Date registerDate;

    @DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")
    private Date orderDate;

    //省略getter/setter
}
```

此处我们使用了Spring字段级别解析/格式化的两个内置注解：

@Number：定义数字相关的解析/格式化元数据（通用样式、货币样式、百分数样式），参数如下：

style：用于指定样式类型，包括三种：Style.NUMBER（通用样式） Style.CURRENCY（货币样式） Style.PERCENT（百分数样式），默认Style.NUMBER；

pattern：自定义样式，如patter="#,###"；

@DateTimeFormat：定义日期相关的解析/格式化元数据，参数如下：

pattern：指定解析/格式化字段数据的模式，如"yyyy-MM-dd HH:mm:ss"

iso：指定解析/格式化字段数据的ISO模式，包括四种：ISO.NONE（不使用） ISO.DATE(yyyy-MM-dd) ISO.TIME(hh:mm:ss.SSSZ) ISO.DATE_TIME(yyyy-MM-dd hh:mm:ss.SSSZ)，默认ISO.NONE；

style：指定用于格式化的样式模式，默认"SS"，具体使用请参考Joda-Time类库的org.joda.time.format.DateTimeFormat的forStyle的javadoc；

优先级： pattern 大于 iso 大于 style。

（**2**、测试用例：

```
@Test
public void test() throws SecurityException, NoSuchFieldException {
    //默认自动注册对@NumberFormat和@DateTimeFormat的支持
    DefaultFormattingConversionService conversionService =
                                new DefaultFormattingConversionService();

    //准备测试模型对象
    FormatterModel model = new FormatterModel();
    model.setTotalCount(10000);
    model.setDiscount(0.51);
    model.setSumMoney(10000.13);
    model.setRegisterDate(new Date(2012-1900, 4, 1));
    model.setOrderDate(new Date(2012-1900, 4, 1, 20, 18, 18));

    //获取类型信息
    TypeDescriptor descriptor =
            new TypeDescriptor(FormatterModel.class.getDeclaredField("totalCount"));
    TypeDescriptor stringDescriptor = TypeDescriptor.valueOf(String.class);

    Assert.assertEquals("10,000", conversionService.convert(model.getTotalCount(), descri
    Assert.assertEquals(model.getTotalCount(), conversionService.convert("10,000", string

}
```

TypeDescriptor：拥有类型信息的上下文，用于Spring3类型转换系统获取类型信息的（可以包含类、字段、方法参数、属性信息）；通过TypeDescriptor，我们就可以获取（类、字段、方法参数、属性）的各种信息，如注解类型信息；

conversionService.convert(model.getTotalCount(), descriptor, stringDescriptor)：将totalCount格式化为字符串类型，此处会根据totalCount字段的注解信息（通过descriptor对象获取）来进行格式化；

conversionService.convert("10,000", stringDescriptor, descriptor)：将字符串“10,000”解析为totalCount字段类型，此处会根据totalCount字段的注解信息（通过descriptor对象获取）来进行解析。

（**3**、通过为不同的字段指定不同的注解信息进行字段级别的细粒度数据解析**/**格式化

```
descriptor = new TypeDescriptor(FormatterModel.class.getDeclaredField("registerDate"));
Assert.assertEquals("2012-05-01", conversionService.convert(model.getRegisterDate(), desc
Assert.assertEquals(model.getRegisterDate(), conversionService.convert("2012-05-01", stri

descriptor = new TypeDescriptor(FormatterModel.class.getDeclaredField("orderDate"));
Assert.assertEquals("2012-05-01 20:18:18", conversionService.convert(model.getOrderDate()
Assert.assertEquals(model.getOrderDate(), conversionService.convert("2012-05-01 20:18:18"
```

通过如上测试可以看出，我们可以通过字段注解方式实现细粒度的数据解析/格式化控制，但是必须使用TypeDescriptor来指定类型的上下文信息，即编程实现字段的数据解析/格式化比较麻烦。

其他测试用例请参考cn.javass.chapter7.web.controller.support.formatter.InnerFieldFormatterTest的test测试方法。

二、自定义注解进行字段级别的解析**/**格式化：

此处以解析/格式化PhoneNumberModel字段为例。

（**1**、定义解析**/**格式化字段的注解类型：

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface PhoneNumber {
}
```

(2、实现**AnnotationFormatterFactory**注解格式化工厂：

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
public class PhoneNumberFormatAnnotationFormatterFactory
    implements AnnotationFormatterFactory<PhoneNumber> {//①指定可以解析/格式化的字段注解类型

    private final Set<Class<?>> fieldTypes;
    private final PhoneNumberFormatter formatter;
    public PhoneNumberFormatAnnotationFormatterFactory() {
        Set<Class<?>> set = new HashSet<Class<?>>();
        set.add(PhoneNumberModel.class);
        this.fieldTypes = set;
        this.formatter = new PhoneNumberFormatter();//此处使用之前定义的Formatter实现
    }
    //②指定可以被解析/格式化的字段类型集合
    @Override
    public Set<Class<?>> getFieldTypes() {
        return fieldTypes;
    }
    //③根据注解信息和字段类型获取解析器
    @Override
    public Parser<?> getParser(PhoneNumber annotation, Class<?> fieldType) {
        return formatter;
    }
    //④根据注解信息和字段类型获取格式化器
    @Override
    public Printer<?> getPrinter(PhoneNumber annotation, Class<?> fieldType) {
        return formatter;
    }
}
```

AnnotationFormatterFactory实现会根据注解信息和字段类型获取相应的解析器/格式化器。

（**3**、修改**FormatterModel**添加如下代码：

```
@PhoneNumber
private PhoneNumberModel phoneNumber;
```

（**4**、测试用例

```
@Test
public void test() throws SecurityException, NoSuchFieldException {
    DefaultFormattingConversionService conversionService =
                                    new DefaultFormattingConversionService();//创建格式
    conversionService.addFormatterForFieldAnnotation(
                    new PhoneNumberFormatAnnotationFormatterFactory());//添加自定义的注解格式

    FormatterModel model = new FormatterModel();
    TypeDescriptor descriptor =
            new TypeDescriptor(FormatterModel.class.getDeclaredField("phoneNumber"));
    TypeDescriptor stringDescriptor = TypeDescriptor.valueOf(String.class);

    PhoneNumberModel value = (PhoneNumberModel) conversionService.convert("010-12345678",
    model.setPhoneNumber(value);

    Assert.assertEquals("010-12345678", conversionService.convert(model.getPhoneNumber(),
}
```

此处使用DefaultFormattingConversionService的addFormatterForFieldAnnotation注册自定义的注解格式化工厂PhoneNumberFormatAnnotationFormatterFactory。

到此，编程进行数据的格式化/解析我们就完成了，使用起来还是比较麻烦，接下来我们将其集成到Spring Web MVC环境中。

7.3.4、集成到Spring Web MVC环境

一、注册**FormattingConversionService**实现和自定义格式化转换器：

```
<bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
  <!—此处省略之前注册的自定义类型转换器-->
  <property name="formatters">
      <list>
           <bean class="cn.javass.chapter7.web.controller.support.formatter.
                                                       PhoneNumberFormatAnnotat
      </list>
  </property>
</bean>
```

其他配置和之前学习7.2.2.4一节一样。

二、示例：

（**1**、模型对象字段的数据解析/格式化：

```
@RequestMapping(value = "/format1")
public String test1(@ModelAttribute("model") FormatterModel formatModel) {
    return "format/success";
}
```

```
totalCount:<spring:bind path="model.totalCount">${status.value}</spring:bind><br/>
discount:<spring:bind path="model.discount">${status.value}</spring:bind><br/>
sumMoney:<spring:bind path="model.sumMoney">${status.value}</spring:bind><br/>
phoneNumber:<spring:bind path="model.phoneNumber">${status.value}</spring:bind><br/>
<!-- 如果没有配置org.springframework.web.servlet.handler.ConversionServiceExposingIntercept
phoneNumber:<spring:eval expression="model.phoneNumber"></spring:eval><br/>

<br/><br/>
<form:form commandName="model">
    <form:input path="phoneNumber"/><br/>
    <form:input path="sumMoney"/>
</form:form>
```

在浏览器输入测试URL：

http://localhost:9080/springmvc-chapter7/format1?totalCount=100000&discount=0.51&sumMoney=100000.128&phoneNumber=010-12345678

数据会正确绑定到我们的formatModel，即请求参数能被正确的解析并绑定到我们的命令对象上，而且在JSP页面也能正确的显示格式化后的数据（即正确的被格式化显示）。

（**2**、功能处理方法参数级别的数据解析：

```
@RequestMapping(value = "/format2")
public String test2(
        @PhoneNumber @RequestParam("phoneNumber") PhoneNumberModel phoneNumber,
        @DateTimeFormat(pattern="yyyy-MM-dd") @RequestParam("date") Date date) {
        System.out.println(phoneNumber);
        System.out.println(date);
        return "format/success2";
}
```

此处我们可以直接在功能处理方法的参数上使用格式化注解类型进行注解，Spring Web MVC能根据此注解信息对请求参数进行解析并正确的绑定。

在浏览器输入测试URL：

http://localhost:9080/springmvc-chapter7/format2?phoneNumber=010-12345678&date=2012-05-01

数据会正确的绑定到我们的phoneNumber和date上，即请求的参数能被正确的解析并绑定到我们的参数上。

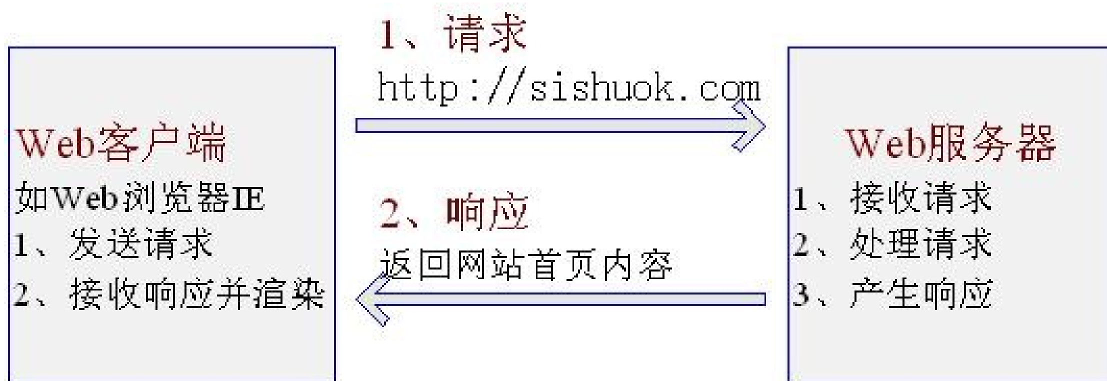
控制器代码位于cn.javass.chapter7.web.controller.DataFormatTestController中。

如果我们请求参数数据不能被正确解析并绑定或输入的数据不合法等该怎么处理呢？接下来的一节我们来学习下绑定失败处理和数据验证相关知识。

第一章 Web MVC简介 —— 跟开涛学SpringMVC

Web MVC简介

1.1、Web开发中的请求-响应模型：



在Web世界里，具体步骤如下：

1、Web浏览器（如IE）发起请求，如访问http://sishuok.com

2、Web服务器（如Tomcat）接收请求，处理请求（比如用户新增，则将把用户保存一下），最后产生响应（一般为html）。

3、web服务器处理完成后，返回内容给web客户端（一般就是我们的浏览器），客户端对接收的内容进行处理（如web浏览器将会对接收到的html内容进行渲染以展示给客户）。

因此，在**Web**世界里：

都是Web客户端发起请求，Web服务器接收、处理并产生响应。

一般Web服务器是不能主动通知Web客户端更新内容。虽然现在有些技术如服务器推（如Comet）、还有现在的HTML5 websocket可以实现Web服务器主动通知Web客户端。

到此我们了解了在web开发时的请求/响应模型，接下来我们看一下标准的MVC模型是什么。

1.2、标准MVC模型概述

MVC模型：是一种架构型的模式，本身不引入新功能，只是帮助我们将开发的结构组织的更加合理，使展示与模型分离、流程控制逻辑、业务逻辑调用与展示逻辑分离。如图1-2

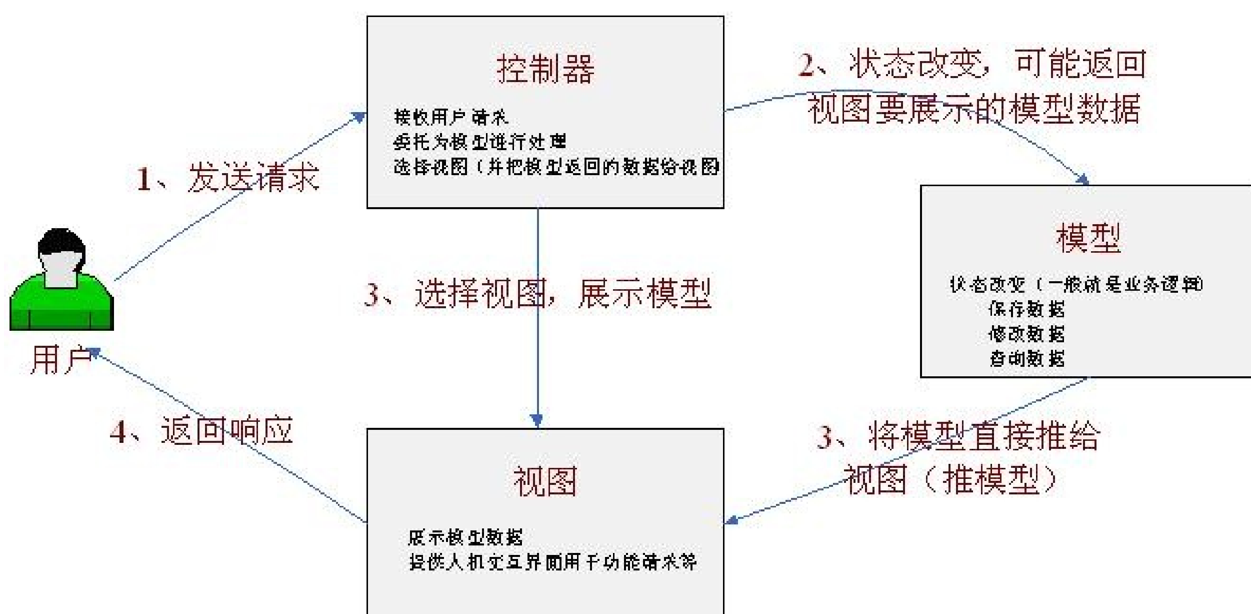


图1-2

首先让我们了解下**MVC**（**Model-View-Controller**）三元组的概念：

Model（模型）：数据模型，提供要展示的数据，因此包含数据和行为，可以认为是领域模型或JavaBean组件（包含数据和行为），不过现在一般都分离开来：Value Object（数据） 和服务层（行为）。也就是模型提供了模型数据查询和模型数据的状态更新等功能，包括数据和业务。

View（视图）：负责进行模型的展示，一般就是我们见到的用户界面，客户想看到的东西。

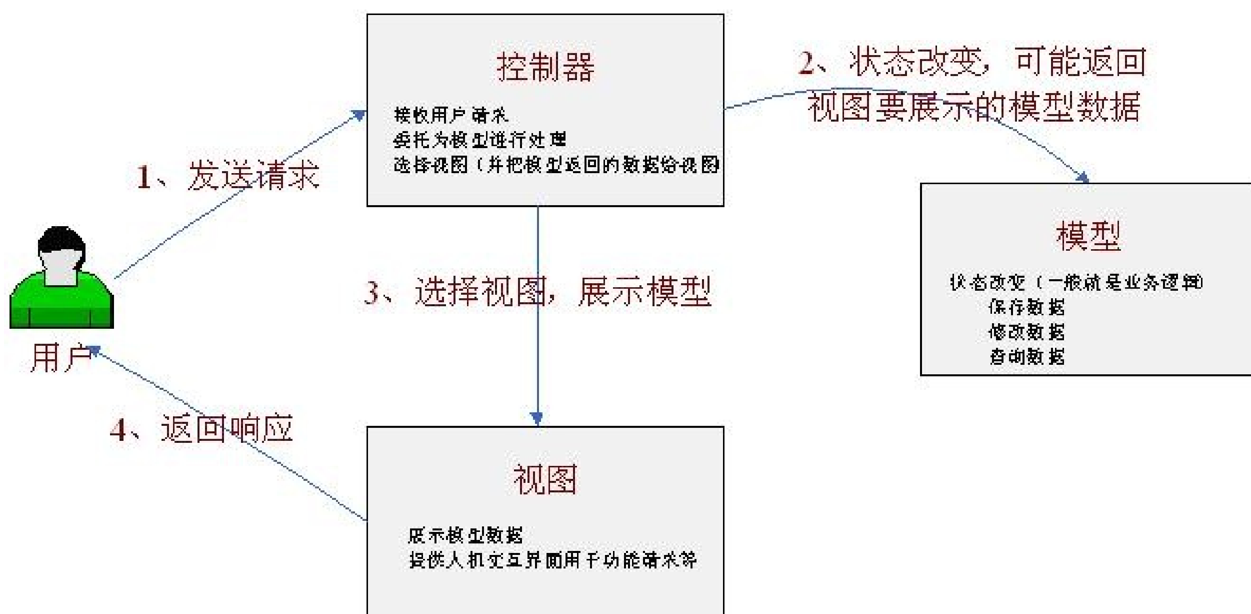
Controller（控制器）：接收用户请求，委托给模型进行处理（状态改变），处理完毕后把返回的模型数据返回给视图，由视图负责展示。 也就是说控制器做了个调度员的工作，。

从图1-1我们还看到，在标准的MVC中模型能主动推数据给视图进行更新（观察者设计模式，在模型上注册视图，当模型更新时自动更新视图），但在Web开发中模型是无法主动推给视图（无法主动更新用户界面），因为在Web开发是请求-响应模型。

那接下来我们看一下在Web里MVC是什么样子，我们称其为 Web MVC 来区别标准的MVC。

1.3、Web MVC概述

模型-视图-控制器概念和标准MVC概念一样，请参考1.2，我们再看一下Web MVC标准架构，如图1-3：



如图1-3

在Web MVC模式下，模型无法主动推数据给视图，如果用户想要视图更新，需要再发送一次请求（即请求-响应模型）。

概念差不多了，我们接下来了解下Web端开发的发展历程，和使用代码来演示一下Web MVC是如何实现的，还有为什么要使用MVC这个模式呢？

1.4、Web端开发发展历程

此处我们只是简单的叙述比较核心的历程，如图1-4

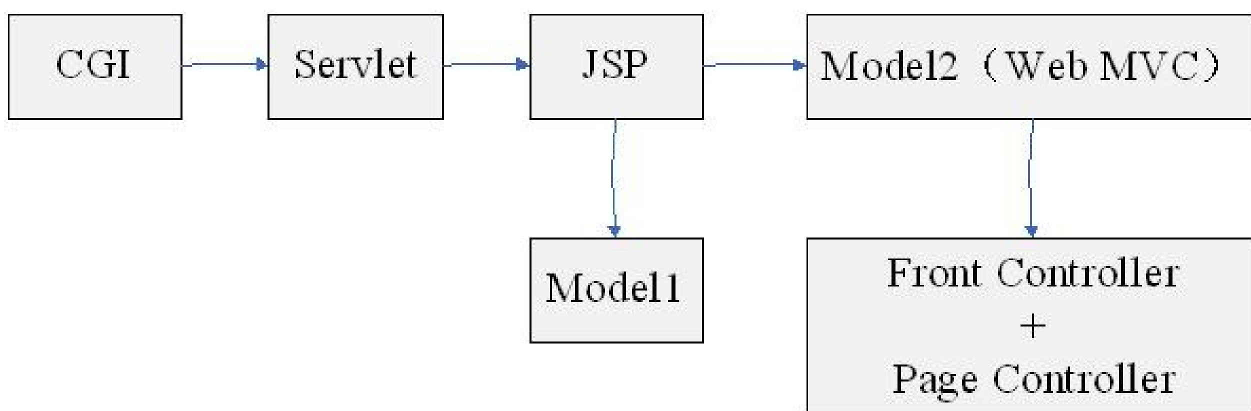


图1-4

1.4.1、CGI：（Common Gateway Interface）公共网关接口，一种在web服务端使用的脚本技术，使用C或Perl语言编写，用于接收web用户请求并处理，最后动态产生响应给用户，但每次请求将产生一个进程，重量级。

1.4.2、Servlet：一种JavaEE web组件技术，是一种在服务器端执行的web组件，用于接收web用户请求并处理，最后动态产生响应给用户。但每次请求只产生一个线程（而且有线程池），轻量级。而且能利用许多JavaEE技术（如JDBC等）。本质就是在里面 输出 html流。但表现逻辑、控制逻辑、业务逻辑调用混杂。如图1-5

```
public class LoginServlet extends HttpServlet {
        @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        doPost(req, resp);//为了简单，直接委托给doPost进行处理
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        String submitFlag = req.getParameter("submitFlag");
        if("toLogin".equals(submitFlag)) {
            toLogin(req, resp);return;
        } else if("login".equals(submitFlag)) {
            login(req, resp);return;
        }
        toLogin(req, resp); //默认到登录页面
    }
    private void toLogin(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
       resp.setContentType("text/html");
       String loginPath = req.getContextPath() + "/servletLogin";
       PrintWriter write = resp.getWriter();
       write.write("<form action='" + loginPath + "' method='post'>");
       write.write("<input type='text' name='submitFlag' value='login'/>");
       write.write("username:<input type='text' name='username'/>");
       write.write("password:<input type='password' name='password'/>");
       write.write("<input type='submit' value='login'/>");
       write.write("</form>");
    }
    private void login(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
       //1收集参数
       String username = req.getParameter("username");
       String password = req.getParameter("password");
       //2验证并封装参数（重复的步骤）
       UserBean user = new UserBean();
       user.setUsername(username);
       user.setPassword(password);
       //3调用javabean对象（业务方法）
       if(user.login()) {
               //4根据返回值选择下一个页面
               resp.getWriter().write("login success");
       } else {
               resp.getWriter().write("login fail");
       }
    }
}
```

1、控制逻辑：根据请求参数选择要执行的功能方法

2、表现代码：页面展示直接放在我们的servlet里边

3、调用业务对象(javabean对象)进行登录：即模型，不仅包含数据还包含行为

图1-5

如图1-5，这种做法是绝对不可取的，控制逻辑、表现代码、业务逻辑对象调用混杂在一起，最大的问题是直接在里面输出Html，这样前端开发人员无法进行页面风格等的设计与修改，即使修改也是很麻烦，因此实际项目这种做法不可取。

1.4.3、JSP：（Java Server Page）：一种在服务器端执行的web组件，是一种运行在标准的HTML页面中嵌入脚本语言（现在只支持Java）的模板页面技术。本质就是在html代码中嵌入。JSP最终还是会被编译为Servlet，只不过比纯Servlet开发页面更简单、方便。但表现逻辑、控制逻辑、业务逻辑调用还是混杂。如图1-6

```
<%@page import="cn.javass.chapter1.javabean.UserBean"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录</title>
</head>
<body>
<%
    String submitFlag = request.getParameter("submitFlag");
    if("login".equals(submitFlag)) {//登录
        //1收集参数
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        //2验证并封装参数
        UserBean user = new UserBean();
        user.setUsername(username);
        user.setPassword(password);
        //33调用javabean对象（业务方法）
        if(user.login()) {
              //4根据返回值选择下一个页面
            out.write("login success");
        } else {
            out.write("login fail");
        }
    } else {
%>
                <form action="" method="post">
                    <input type="hidden" name="submitFlag" value="login"/>

                    username:<input type="text" name="username"/><br/>
                    password:<input type="password" name="password"/><br/>
                    <input type="submit" value="login"/>
                </form>

<%          } %>

</body>
</html>
```

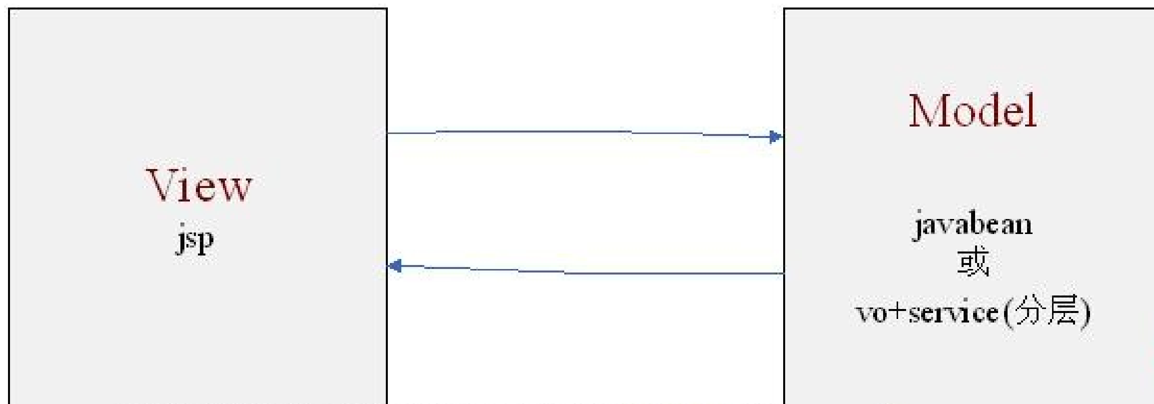
1、控制逻辑：根据请求参数选择要执行的功能方法

3、调用业务对象(javabean对象)进行登录：即模型，不仅包含数据还包含行为

2、表现代码：页面展示直接放在我们的servlet里边

图1-6

如图1-6，这种做法也是绝对不可取的，控制逻辑、表现代码、业务逻辑对象调用混杂在一起，但比直接在servlet里输出html要好一点，前端开发人员可以进行简单的页面风格等的设计与修改（但如果嵌入的java脚本太多也是很难修改的），因此实际项目这种做法不可取。



JSP本质还是Servlet，最终在运行时会生成一个Servlet（如tomcat，将在tomcat\work\Catalina\web应用名\org\apache\jsp下生成），但这种使得写html简单点，但仍是控制逻辑、表现代码、业务逻辑对象调用混杂在一起。

1.4.4、Model1：可以认为是JSP的增强版，可以认为是jsp+javabean如图1-7

特点：使用<jsp:useBean>标准动作，自动将请求参数封装为JavaBean组件；还必须使用java脚本执行控制逻辑。

```
<%@page import="cn.javass.chapter1.javabean.UserBean"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
//1收集参数
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        //2验证并封装参数
        UserBean user = new UserBean();
        user.setUsername(username);
        user.setPassword(password);
" "http://www.w3.org/TR/html4/loose.dtd">
TF-8">

<%-- 创建javabean --%>
<jsp:useBean id="user" class="cn.javass.chapter1.javabean.UserBean"/>
<%--1、收集参数并封装参数（比直接使用jsp，在这块是简单的） --%>
<jsp:setProperty name="user" property="*"/>

<%
    String submitFlag = request.getParameter("submitFlag");
    if("login".equals(submitFlag)) {//登录
        //3调用javabean对象（业务方法）
        if(user.login()) {
            //4根据返回值选择下一个页面
            out.write("login success");
        } else {
            out.write("login fail");
        }
    } else {
%>
            <form action="" method="post">
                <input type="hidden" name="submitFlag" value="login"/>
                username:<input type="text" name="username"/><br/>
                password:<input type="password" name="password"/><br/>
                <input type="submit" value="login"/>
            </form>
<%       } %>

</body>
</html>
```

收集参数和组织参数简单了许多

1、控制逻辑：根据请求参数选择要执行的功能方法

3、调用业务对象(javabean对象)进行登录：即模型，不仅包含数据还包含行为

2、表现代码：页面展示直接放在我们的servlet里边

图1-7

此处我们可以看出，使用<jsp:useBean>标准动作可以简化javabean的获取/创建，及将请求参数封装到javabean，再看一下Model1架构，如图1-8。

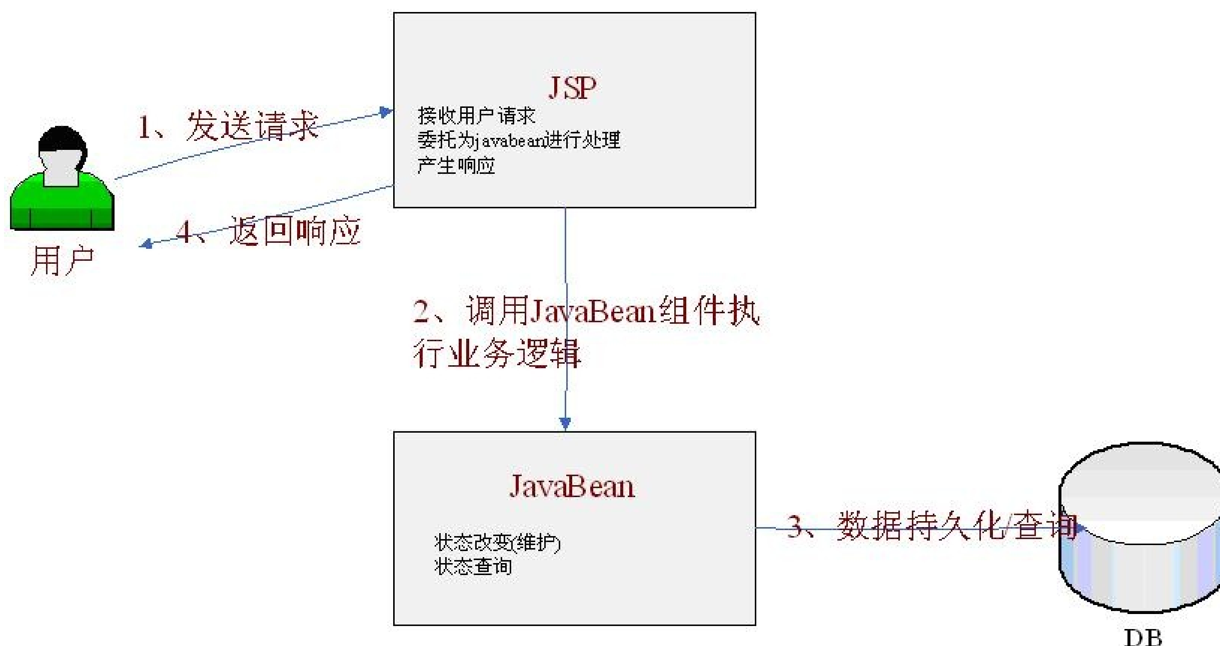


图1-8 Model1架构

Model1架构中，JSP负责控制逻辑、表现逻辑、业务对象（javabean）的调用，只是比纯JSP简化了获取请求参数和封装请求参数。同样是不好的，在项目中应该严禁使用（或最多再demo里使用）。

1.4.5、Model2：在JavaEE世界里，它可以认为就是Web MVC模型

Model2架构其实可以认为就是我们所说的Web MVC模型，只是控制器采用Servlet、模型采用JavaBean、视图采用JSP，如图1-9

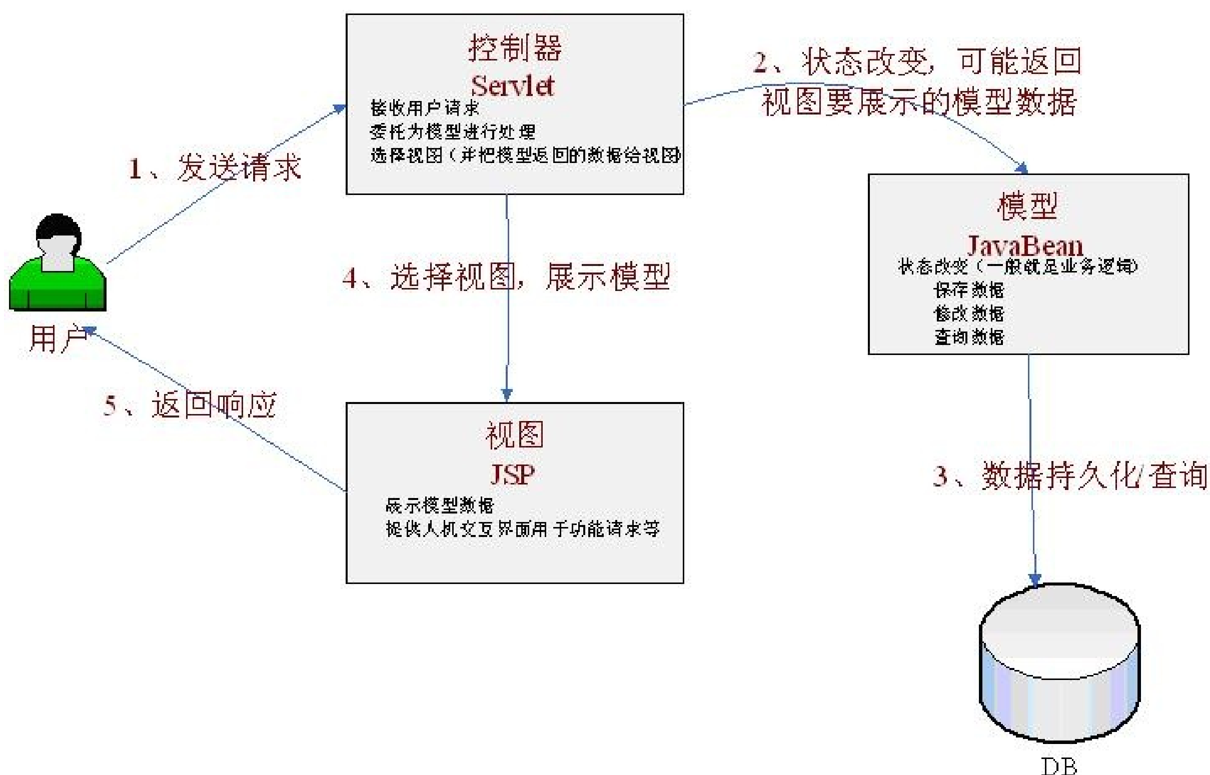


图1-9 Model2架构

具体代码事例如下：

模型

```java
public class UserBean implements java.io.Serializable {

	private String username;
	private String password;
	public String getUsername() {
		return username;
	}
	public void setUsername(String username) {
		this.username = username;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}

	/**
	 * 因为我们只关注表现层，此处只是模拟，实际项目需要改掉！
	 * @param username 用户名
	 * @param password 密码
	 * @return
	 */
	public boolean login() {
		if("zhang".equals(this.username) && "123".equals(this.password)) {
			return true;
		}
		return false;
	}

}
```

模型: (业务对象 JavaBean对象)
包含设置/获取数据的方法
包含业务方法

视图

```jsp
<%@page import="cn.javass.chapter1.javabean.UserBean"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录</title>
</head>
<body>

<form action="${pageContext.request.contextPath}/model2Login" method="post">
    <input type="hidden" name="submitFlag" value="login"/>
    username:<input type="text" name="username" value="${user.username}"/><br/>
    password:<input type="password" name="password"/><br/>
    <input type="submit" value="login"/>
</form>

</body>
</html>
```

模型展示,可能包含一些展示逻辑

展示逻辑如在网站首页
如果用户已登陆,显示”欢迎访问,sishuok”
如果用户未登陆,显示”欢迎访问,游客”

控制器

```java
public class Model2Servlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        doPost(req, resp);//为了简单，直接委托给doPost进行处理
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        String submitFlag = req.getParameter("submitFlag");
        if("toLogin".equals(submitFlag)) {
            toLogin(req, resp);
            return;
        } else if("login".equals(submitFlag)) {
            login(req, resp);
            return;
        }
        toLogin(req, resp); //默认到登录页面
    }
    private void toLogin(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        //此处和JSP视图技术紧密耦合，更换其他视图技术几乎不可能
        req.getRequestDispatcher("/mvc/login.jsp").forward(req, resp);
    }
    private void login(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
        //1收集参数
        String username = req.getParameter("username");
        String password = req.getParameter("password");
        //2验证并封装参数（重复的步骤）
        UserBean user = new UserBean();
        user.setUsername(username);
        user.setPassword(password);
        //3调用javabean对象（业务方法）
        if(user.login()) {
                //4根据返回值选择下一个页面
                resp.sendRedirect(req.getContextPath() + "mvc/success.jsp");
        } else {
                //登陆失败再返回登陆页面  并显示上次输入的用户名

                //将视图要显示的模型数据放在请求里传递给视图 ，视图再来展示
                //此处也可以看出和Servlet API紧密耦合，更换视图技术也要重新设置这些数据
                req.setAttribute("user", user);
                toLogin(req, resp);
                return;
        }
    }
}
```

1、控制逻辑：根据请求参数选择要执行的功能方法

2、调用业务对象(javabean对象)进行登录：即模型，不仅包含数据还包含行为

3、选择下一个视图进行模型展示，模型数据直接放在request里

从Model2架构可以看出，视图和模型分离了，控制逻辑和展示逻辑分离了。

但我们也看到严重的缺点：

1．1、控制器：

1．1．1、控制逻辑可能比较复杂，其实我们可以按照规约，如请求参数submitFlag=toAdd，我们其实可以直接调用toAdd方法，来简化控制逻辑；而且每个模块基本需要一个控制器，造成控制逻辑可能很复杂；

1．1．2、请求参数到模型的封装比较麻烦，如果能交给框架来做这件事情，我们可以从中得到解放；

1．1．3、选择下一个视图，严重依赖Servlet API，这样很难或基本不可能更换视图；

1．1．4、给视图传输要展示的模型数据，使用Servlet API，更换视图技术也要一起更换，很麻烦。

1.2、模型：

1．2．1、此处模型使用JavaBean，可能造成JavaBean组件类很庞大，一般现在项目都是采用三层架构，而不采用JavaBean。



JavaBean组件 等价于 域模型层+业务逻辑层+持久层

1.3、视图

1·3·1、现在被绑定在JSP，很难更换视图，比如Velocity、FreeMarker；比如我要支持Excel、PDF视图等等。

1.4.5、服务到工作者：**Front Controller + Application Controller + Page Controller + Context**

即，前端控制器+应用控制器+页面控制器（也有称其为动作）+上下文，也是Web MVC，只是责任更加明确，详情请参考《核心J2EE设计模式》和《企业应用架构模式》如图1-10：

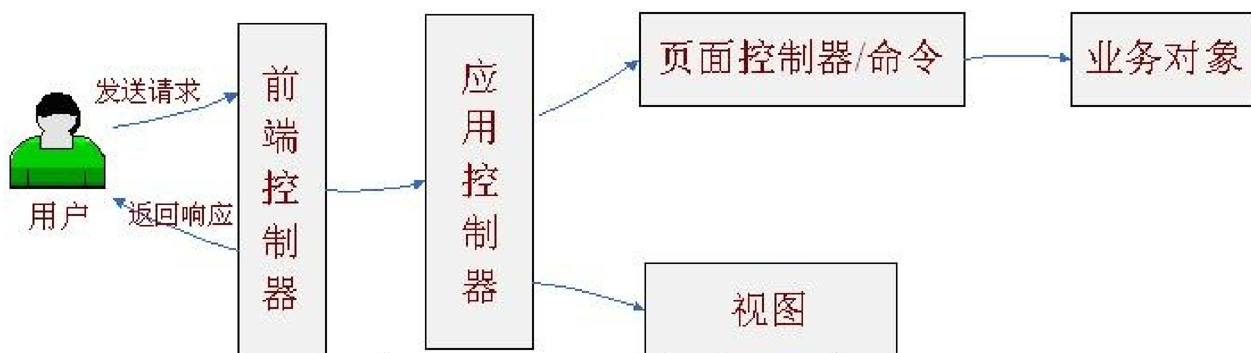
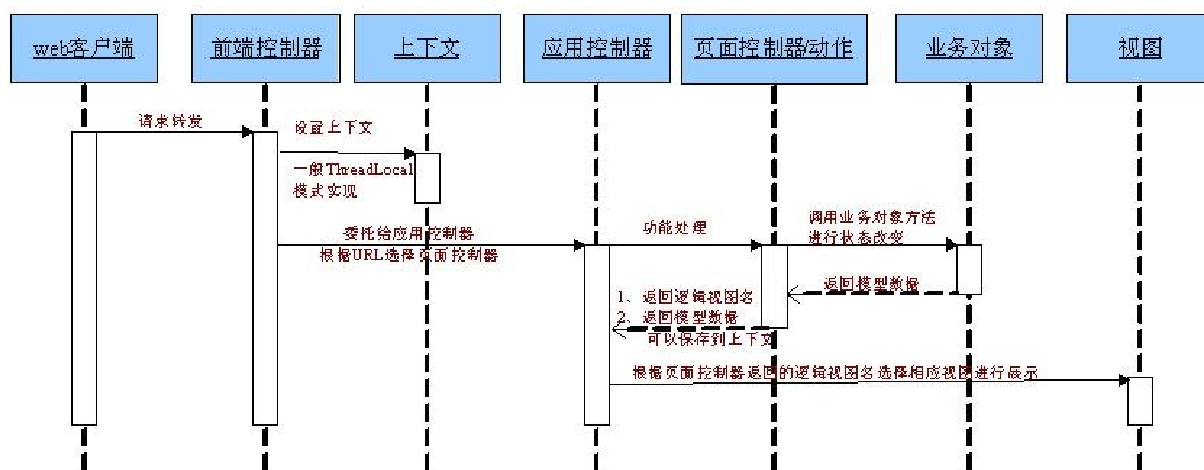


图1-10

运行流程如下：



职责：

Front Controller：前端控制器，负责为表现层提供统一访问点，从而避免Model2中出现的重复的控制逻辑（由前端控制器统一回调相应的功能方法，如前边的根据submitFlag=login转调login方法）；并且可以为多个请求提供共用的逻辑（如准备上下文等等），将选择具体视图和具体的功能处理（如login里边封装请求参数到模型，并调用业务逻辑对象）分离。

Application Controller：应用控制器，前端控制器分离选择具体视图和具体的功能处理之后，需要有人来管理，应用控制器就是用来选择具体视图技术（视图的管理）和具体的功能处理（页面控制器/命令对象/动作管理），一种策略设计模式的应用，可以很容易的切换视图/页面控制器，相互不产生影响。

Page Controller(Command)：页面控制器/动作/处理器：功能处理代码，收集参数、封装参数到模型，转调业务对象处理模型，返回逻辑视图名交给前端控制器（和具体的视图技术解耦），由前端控制器委托给应用控制器选择具体的视图来展示，可以是命令设计模式的实现。页面控制器也被称为处理器或动作。

Context：上下文，还记得Model2中为视图准备要展示的模型数据吗，我们直接放在request中（Servlet API相关），有了上下文之后，我们就可以将相关数据放置在上下文，从而与协议无关（如Servlet API）的访问/设置模型数据，一般通过ThreadLocal模式实现。

到此，我们回顾了整个web开发架构的发展历程，可能不同的web层框架在细节处理方面不同，但的目的是一样的：

干净的web表现层：

模型和视图的分离；

控制器中的控制逻辑与功能处理分离（收集并封装参数到模型对象、业务对象调用）；

控制器中的视图选择与具体视图技术分离。

轻薄的web表现层：

做的事情越少越好，薄薄的，不应该包含无关代码；

只负责收集并组织参数到模型对象，启动业务对象的调用；

控制器只返回逻辑视图名并由相应的应用控制器来选择具体使用的视图策略；

尽量少使用框架特定API，保证容易测试。

到此我们了解Web MVC的发展历程，接下来让我们了解下Spring MVC到底是什么、架构及来个HelloWorld了解下具体怎么使用吧。

本章具体代码请参考 springmvc-chapter1工程。

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

2.1、Spring Web MVC是什么

Spring Web MVC是一种基于Java的实现了Web MVC设计模式的请求驱动类型的轻量级Web框架，即使用了MVC架构模式的思想，将web层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC也是要简化我们日常Web开发的。

另外还有一种基于组件的、事件驱动的Web框架在此就不介绍了，如Tapestry、JSF等。

Spring Web MVC也是服务到工作者模式的实现，但进行可优化。前端控制器是`DispatcherServlet`；应用控制器其实拆为处理器映射器(Handler Mapping)进行处理器管理和视图解析器(View Resolver)进行视图管理；页面控制器/动作/处理器为Controller接口（仅包含`ModelAndView handleRequest(request, response)`方法）的实现（也可以是任何的POJO类）；支持本地化（Locale）解析、主题（Theme）解析及文件上传等；提供了非常灵活的数据验证、格式化和数据绑定机制；提供了强大的约定大于配置（惯例优先原则）的契约式编程支持。

2.2、Spring Web MVC能帮我们做什么

√让我们能非常简单的设计出干净的Web层和薄薄的Web层；

√进行更简洁的Web层的开发；

√天生与Spring框架集成（如IoC容器、AOP等）；

√提供强大的约定大于配置的契约式编程支持；

√能简单的进行Web层的单元测试；

√支持灵活的URL到页面控制器的映射；

√非常容易与其他视图技术集成，如Velocity、FreeMarker等等，因为模型数据不放在特定的API里，而是放在一个Model里（`Map`数据结构实现，因此很容易被其他框架使用）；

√非常灵活的数据验证、格式化和数据绑定机制，能使用任何对象进行数据绑定，不必实现特定框架的API；

√提供一套强大的JSP标签库，简化JSP开发；

√支持灵活的本地化、主题等解析；

√更加简单的异常处理；

√对静态资源的支持；

√支持Restful风格。

2．3、Spring Web MVC架构

Spring Web MVC框架也是一个基于请求驱动的Web框架，并且也使用了前端控制器模式来进行设计，再根据请求映射规则分发给相应的页面控制器（动作/处理器）进行处理。首先让我们整体看一下Spring Web MVC处理请求的流程：

2.3.1、Spring Web MVC处理请求的流程

如图2-1

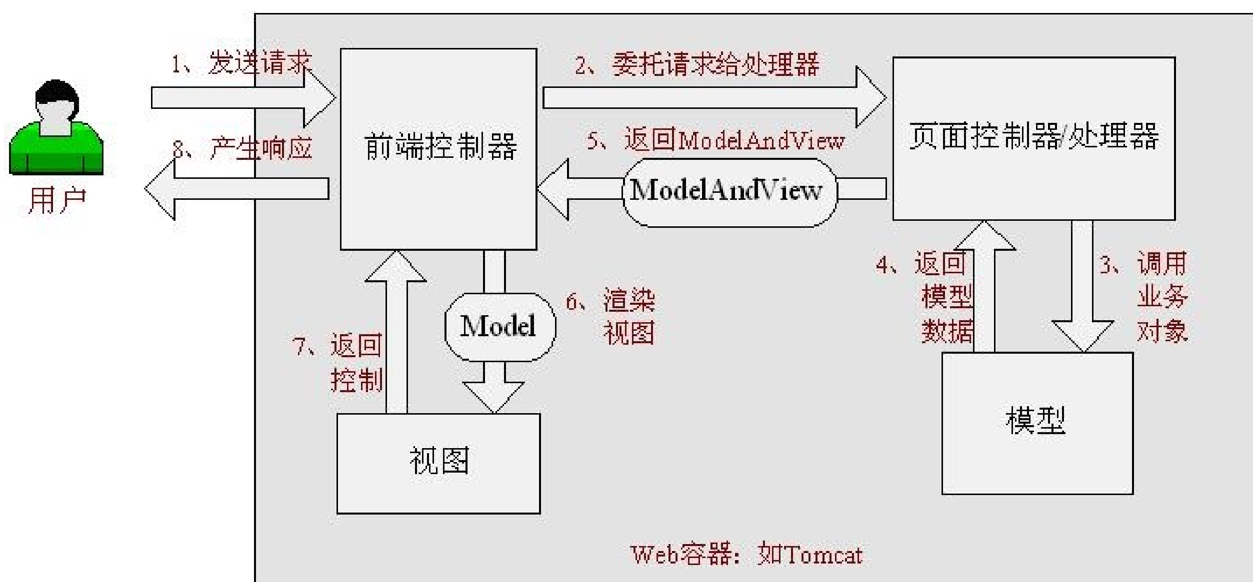


图2-1

具体执行步骤如下：

1、 首先用户发送请求————>前端控制器，前端控制器根据请求信息（如URL）来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；图2-1中的1、2步骤；

2、 页面控制器接收到请求后，进行功能处理，首先需要收集和绑定请求参数到一个对象，这个对象在Spring Web MVC中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个ModelAndView（模型数据和逻辑视图名）；图2-1中的3、4、5步骤；

3、 前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；图2-1中的步骤6、7；

4、 前端控制器再次收回控制权，将响应返回给用户，图2-1中的步骤8；至此整个结束。

问题：

1、 请求如何给前端控制器？

2、 前端控制器如何根据请求信息选择页面控制器进行功能处理？

3、 如何支持多种页面控制器呢？

4、 如何页面控制器如何使用业务对象？

5、 页面控制器如何返回模型数据？

6、 前端控制器如何根据页面控制器返回的逻辑视图名选择具体的视图进行渲染？

7、 不同的视图技术如何使用相应的模型数据？

首先我们知道有如上问题，那这些问题如何解决呢？请让我们先继续，在后边依次回答。

2.3.2、Spring Web MVC架构

1、Spring Web MVC核心架构图，如图2-2

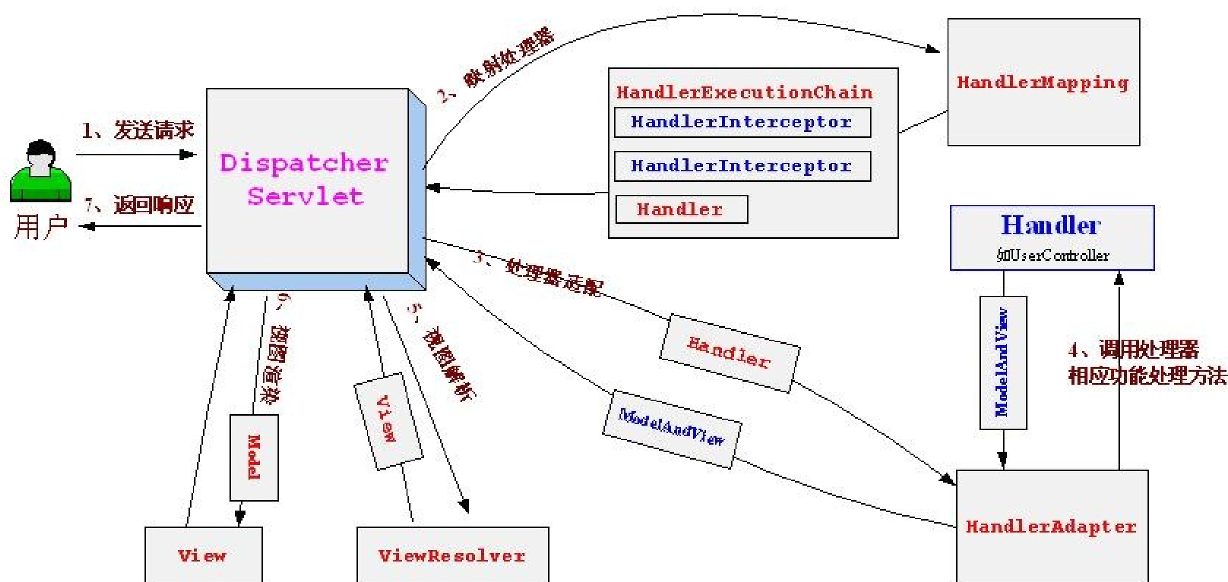


图2-2

架构图对应的DispatcherServlet核心代码如下：

```
//前端控制器分派方法
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throw
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
```

```
        int interceptorIndex = -1;

        try {
            ModelAndView mv;
            boolean errorView = false;

            try {
                   //检查是否是请求是否是multipart（如文件上传），如果是将通过MultipartResolver解
                processedRequest = checkMultipart(request);
                   //步骤2、请求到处理器（页面控制器）的映射，通过HandlerMapping进行映射
                mappedHandler = getHandler(processedRequest, false);
                if (mappedHandler == null || mappedHandler.getHandler() == null) {
                    noHandlerFound(processedRequest, response);
                    return;
                }
                   //步骤3、处理器适配，即将我们的处理器包装成相应的适配器（从而支持多种类型的处理器）
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                  // 304 Not Modified缓存支持
                //此处省略具体代码

                // 执行处理器相关的拦截器的预处理（HandlerInterceptor.preHandle）
                //此处省略具体代码

                // 步骤4、由适配器执行处理器（调用处理器相应功能处理方法）
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                // Do we need view name translation?
                if (mv != null && !mv.hasView()) {
                    mv.setViewName(getDefaultViewName(request));
                }

                // 执行处理器相关的拦截器的后处理（HandlerInterceptor.postHandle）
                //此处省略具体代码
            }
            catch (ModelAndViewDefiningException ex) {
                logger.debug("ModelAndViewDefiningException encountered", ex);
                mv = ex.getModelAndView();
            }
            catch (Exception ex) {
                Object handler = (mappedHandler != null ? mappedHandler.getHandler() : nu
                mv = processHandlerException(processedRequest, response, handler, ex);
                errorView = (mv != null);
            }

            //步骤5 步骤6、解析视图并进行视图的渲染
  //步骤5 由ViewResolver解析View（viewResolver.resolveViewName(viewName, locale)）
  //步骤6 视图在渲染时会把Model传入（view.render(mv.getModelInternal(), request, response);）
            if (mv != null && !mv.wasCleared()) {
                render(mv, processedRequest, response);
                if (errorView) {
                    WebUtils.clearErrorRequestAttributes(request);
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Null ModelAndView returned to DispatcherServlet with na
                            "': assuming HandlerAdapter completed request handling");
                }
            }

            // 执行处理器相关的拦截器的完成后处理（HandlerInterceptor.afterCompletion）
            //此处省略具体代码

        catch (Exception ex) {
            // Trigger after-completion for thrown exception.
            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, res
            throw ex;
        }
        catch (Error err) {
            ServletException ex = new NestedServletException("Handler processing failed",
            // Trigger after-completion for thrown exception.
```

```
            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, res
            throw ex;
        }

        finally {
            // Clean up any resources used by a multipart request.
            if (processedRequest != request) {
                cleanupMultipart(processedRequest);
            }
        }
    }
```

核心架构的具体流程步骤如下：

1、 首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；

2、 DispatcherServlet——>HandlerMapping， HandlerMapping将会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略；

3、 DispatcherServlet——>HandlerAdapter，HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；

4、 HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个ModelAndView对象（包含模型数据、逻辑视图名）；

5、 ModelAndView的逻辑视图名——> ViewResolver， ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；

6、 View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构，因此很容易支持其他视图技术；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

此处我们只是讲了核心流程，没有考虑拦截器、本地解析、文件上传解析等，后边再细述。

到此，再来看我们前边提出的问题：

1、 请求如何给前端控制器？这个应该在web.xml中进行部署描述，在HelloWorld中详细讲解。

2、 前端控制器如何根据请求信息选择页面控制器进行功能处理？ 我们需要配置HandlerMapping进行映射

3、 如何支持多种页面控制器呢？配置HandlerAdapter从而支持多种类型的页面控制器

4、 如何页面控制器如何使用业务对象？可以预料到，肯定利用Spring IoC容器的依赖注入功能

5、 页面控制器如何返回模型数据？使用ModelAndView返回

6、 前端控制器如何根据页面控制器返回的逻辑视图名选择具体的视图进行渲染？ 使用ViewResolver进行解析

7、 不同的视图技术如何使用相应的模型数据？ 因为Model是一个Map数据结构，很容易支持其他视图技术

在此我们可以看出具体的核心开发步骤：

1、 DispatcherServlet在web.xml中的部署描述，从而拦截请求到Spring Web MVC

2、 HandlerMapping的配置，从而将请求映射到处理器

3、 HandlerAdapter的配置，从而支持多种类型的处理器

4、 ViewResolver的配置，从而将逻辑视图名解析为具体视图技术

5、处理器（页面控制器）的配置，从而进行功能处理

上边的开发步骤我们会在Hello World中详细验证。

2·4、Spring Web MVC优势

1、清晰的角色划分：前端控制器（ `DispatcherServlet` ）、请求到处理器映射（HandlerMapping）、处理器适配器（HandlerAdapter）、视图解析器（ViewResolver）、处理器或页面控制器（Controller）、验证器（ Validator）、命令对象（Command 请求参数绑定到的对象就叫命令对象）、表单对象（Form Object 提供给表单展示和提交到的对象就叫表单对象）。

2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要；

3、由于命令对象就是一个POJO，无需继承框架特定API，可以使用命令对象直接作为业务对象；

4、和Spring 其他框架无缝集成，是其它Web框架所不具备的；

5、可适配，通过HandlerAdapter可以支持任意的类作为处理器；

6、可定制性，HandlerMapping、ViewResolver等能够非常简单的定制；

7、功能强大的数据验证、格式化、绑定机制；

8、利用Spring提供的Mock对象能够非常简单的进行Web层单元测试；

9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。

10、强大的JSP标签库，使JSP编写更容易。

………………还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配置支持等等。

到此我们已经简单的了解了Spring Web MVC，接下来让我们来个实例来具体使用下这个框架。

2·5、Hello World入门

2.5.1、准备开发环境和运行环境：

☆开发工具：**eclipse**

☆运行环境：**tomcat6.0.20**

☆工程：动态**web**工程（**springmvc-chapter2**）

☆**spring**框架下载：

spring-framework-3.1.1.RELEASE-with-docs.zip

☆依赖**jar**包：

1、 Spring框架jar包：

为了简单，将spring-framework-3.1.1.RELEASE-with-docs.zip/dist/下的所有jar包拷贝到项目的WEB-INF/lib目录下；

2、 Spring框架依赖的jar包：

需要添加Apache commons logging日志，此处使用的是commons.logging-1.1.1.jar；

需要添加jstl标签库支持，此处使用的是jstl-1.1.2.jar和standard-1.1.2.jar；

2.5.2、前端控制器的配置

在我们的web.xml中添加如下配置：

```
    <servlet>
        <servlet-name>chapter2</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>chapter2</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

load-on-startup：表示启动容器时初始化该Servlet；

url-pattern：表示哪些请求交给Spring Web MVC处理，"/" 是用来定义默认servlet映射的。也可以如"*.html"表示拦截所有以html为扩展名的请求。

自此请求已交给Spring Web MVC框架处理，因此我们需要配置Spring的配置文件，默认DispatcherServlet会加载WEB-INF/[DispatcherServlet的Servlet名字]-servlet.xml配置文件。本示例为WEB-INF/ chapter2-servlet.xml。

2.5.3、在Spring配置文件中配置HandlerMapping、HandlerAdapter

具体配置在WEB-INF/ chapter2-servlet.xml文件中：

```
<!-- HandlerMapping -->
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<!-- HandlerAdapter -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
```

BeanNameUrlHandlerMapping：表示将请求的URL和Bean名字映射，如URL为 "上下文/hello"，则Spring配置文件必须有一个名字为"/hello"的Bean，上下文默认忽略。

SimpleControllerHandlerAdapter：表示所有实现了org.springframework.web.servlet.mvc.Controller接口的Bean可以作为Spring Web MVC中的处理器。如果需要其他类型的处理器可以通过实现HadlerAdapter来解决。

2.5.4、在Spring配置文件中配置ViewResolver

具体配置在WEB-INF/ chapter2-servlet.xml文件中：

```
<!-- ViewResolver -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

InternalResourceViewResolver：用于支持Servlet、JSP视图解析；

viewClass：JstlView表示JSP模板页面需要使用JSTL标签库，classpath中必须包含jstl的相关jar包；

prefix和suffix：查找视图页面的前缀和后缀（前缀[逻辑视图名]后缀），比如传进来的逻辑视图名为hello，则该该jsp视图页面应该存放在“WEB-INF/jsp/hello.jsp”；

2.5.5、开发处理器/页面控制器

```
package cn.javass.chapter2.web.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
public class HelloWorldController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) t
       //1、收集参数、验证参数
       //2、绑定参数到命令对象
       //3、将命令对象传入业务对象进行业务处理
       //4、选择下一个页面
       ModelAndView mv = new ModelAndView();
       //添加模型数据 可以是任意的POJO对象
       mv.addObject("message", "Hello World!");
       //设置逻辑视图名，视图解析器会根据该名字解析到具体的视图页面
       mv.setViewName("hello");
       return mv;
    }
}
```

org.springframework.web.servlet.mvc.Controller：页面控制器/处理器必须实现Controller接口，注意别选错了；后边我们会学习其他的处理器实现方式；

public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp)：功能处理方法，实现相应的功能处理，比如收集参数、验证参数、绑定参数到命令对象、将命令对象传入业务对象进行业务处理、最后返回ModelAndView对象；

ModelAndView：包含了视图要实现的模型数据和逻辑视图名；“mv.addObject("message", "Hello World!");

”表示添加模型数据，此处可以是任意POJO对象；“mv.setViewName("hello");”表示设置逻辑视图名为“hello”，视图解析器会将其解析为具体的视图，如前边的视图解析器InternalResourceVi。wResolver会将其解析为“WEB-INF/jsp/hello.jsp”。

我们需要将其添加到Spring配置文件(WEB-INF/chapter2-servlet.xml)，让其接受Spring IoC容器管理:

```
<!-- 处理器 -->
<bean name="/hello" class="cn.javass.chapter2.web.controller.HelloWorldController"/>
```

name="/hello"：前边配置的BeanNameUrlHandlerMapping，表示如过请求的URL为 “上下文/hello”，则将会交给该Bean进行处理。

2.5.6、开发视图页面

创建 /WEB-INF/jsp/hello.jsp视图页面：

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Hello World</title>
</head>
<body>
${message}
</body>
</html>
```

${message}：表示显示由HelloWorldController处理器传过来的模型数据。

2.5.6、启动服务器运行测试

通过请求：http://localhost:9080/springmvc-chapter2/hello，如果页面输出“Hello World! ”就表明我们成功了！

2.5.7、运行流程分析

如图2-3

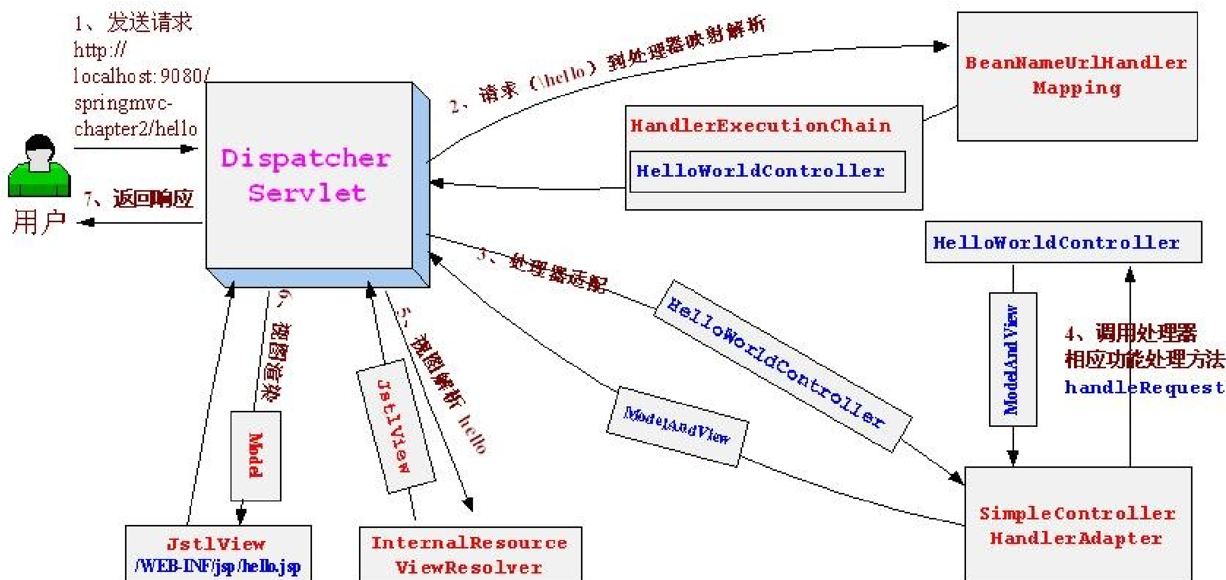


图2-3

运行步骤：

1、首先用户发送请求http://localhost:9080/springmvc-chapter2/hello——>web容器，web容器根据“/hello”路径映射到DispatcherServlet（url-pattern为/）进行处理；

2、DispatcherServlet——>BeanNameUrlHandlerMapping进行请求到处理的映射，BeanNameUrlHandlerMapping将“/hello”路径直接映射到名字为“/hello”的Bean进行处理，即HelloWorldController，BeanNameUrlHandlerMapping将其包装为HandlerExecutionChain（只包括HelloWorldController处理器，没有拦截器）；

3、DispatcherServlet——> SimpleControllerHandlerAdapter，SimpleControllerHandlerAdapter将HandlerExecutionChain中的处理器（HelloWorldController）适配为SimpleControllerHandlerAdapter；

4、SimpleControllerHandlerAdapter——> HelloWorldController处理器功能处理方法的调用，SimpleControllerHandlerAdapter将会调用处理器的handleRequest方法进行功能处理，该处理方法返回一个ModelAndView给DispatcherServlet；

5、hello（ModelAndView的逻辑视图名）——>InternalResourceViewResolver，InternalResourceViewResolver使用JstlView，具体视图页面在/WEB-INF/jsp/hello.jsp；

6、JstlView（/WEB-INF/jsp/hello.jsp）——>渲染，将在处理器传入的模型数据(message=HelloWorld！)在视图中展示出来；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

到此HelloWorld就完成了，步骤是不是有点多？而且回忆下我们主要进行了如下配置：

1、前端控制器DispatcherServlet；

2、HandlerMapping

3、HandlerAdapter

4、ViewResolver

5、处理器/页面控制器

6、视图

因此，接下来几章让我们详细看看这些配置，先从DispatcherServlet开始吧。

2.6、POST中文乱码解决方案

spring Web MVC框架提供了org.springframework.web.filter.CharacterEncodingFilter用于解决POST方式造成的中文乱码问题，具体配置如下：

```
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

以后我们项目及所有页面的编码均为UTF-8。

2.7、Spring3.1新特性

`**一、Spring2.5之前，我们都是通过实现Controller接口或其实现来定义我们的处理器类。**`

`**二、Spring2.5引入注解式处理器支持，通过@Controller 和 @RequestMapping注解定义我们的处理器类。并且提`

`需要通过处理器映射DefaultAnnotationHandlerMapping和处理器适配器AnnotationMethodHandlerAdapter来开`

`@Controller：``用于标识是处理器类；`

`@RequestMapping：``请求到处理器功能方法的映射规则；`

`@RequestParam：``请求参数到处理器功能处理方法的方法参数上的绑定；`

`@ModelAttribute：``请求参数到命令对象的绑定；`

`@SessionAttributes：``用于声明session级别存储的属性，放置在处理器类上，通常列出模型属性（如@ModelAttr`

`@InitBinder：``自定义数据绑定注册支持，用于将请求参数转换到命令对象属性的对应类型；`

`**三、Spring3.0引入RESTful架构风格支持(通过@PathVariable注解和一些其他特性支持),且又引入了更多的注解支`

`@CookieValue：``cookie数据到处理器功能处理方法的方法参数上的绑定；`

`@RequestHeader：``请求头（header）数据到处理器功能处理方法的方法参数上的绑定；`

`@RequestBody：``请求的body体的绑定（通过HttpMessageConverter进行类型转换）；`

`@ResponseBody：``处理器功能处理方法的返回值作为响应体（通过HttpMessageConverter进行类型转换）；`

`@ResponseStatus：``定义处理器功能处理方法/异常处理器返回的状态码和原因；`

`@ExceptionHandler：``注解式声明异常处理器；`

`@PathVariable：``请求URI中的模板变量部分到处理器功能处理方法的方法参数上的绑定，从而支持RESTful架构风格`

`**四、还有比如：**`

`JSR-303验证框架的无缝支持（通过@Valid注解定义验证元数据）；`

`使用Spring 3开始的ConversionService进行类型转换（` PropertyEditor依然有效 `），支持` 使用 @NumberFormat 和 @DateTimeFormat来进行数字和日期的格式化；

HttpMessageConverter``（Http输入/输出转换器，比如JSON、XML等的数据输出转换器）；

ContentNegotiatingViewResolver``，内容协商视图解析器，它还是视图解析器，只是它支持根据请求信息将同一模

Spring 3 引入 一个 mvc XML的命名空间用于支持mvc配置，包括如：

<mvc:annotation-driven>：

自动注册基于注解风格的处理器需要的DefaultAnnotationHandlerMapping、AnnotationMethodHandlerAdapte

支持Spring3的ConversionService自动注册

支持JSR-303验证框架的自动探测并注册（只需把JSR-303实现放置到classpath）

自动注册相应的HttpMessageConverter（用于支持@RequestBody 和 @ResponseBody）（如XML输入输出转换器（

<mvc:interceptors>：注册自定义的处理器拦截器；

<mvc:view-controller>：和ParameterizableViewController类似，收到相应请求后直接选择相应的视图

<mvc:resources>：逻辑静态资源路径到物理静态资源路径的支持；

<mvc:default-servlet-handler>：当在web.xml 中DispatcherServlet使用<url-pattern>/</url-pattern> 映射时，能映射静态资源（当Spring Web MVC框架没有处理请求对应的控制器时（如一些静态资源），转交给默认的Servlet来响应静态文件，否则报404找不到资源错误，）。

……等等。

五、Spring3.1新特性：

对Servlet 3.0的全面支持。

@EnableWebMvc：``用于在基于Java类定义Bean配置中开启MVC支持，和XML中的<mvc:annotation-driven&gt

新的@Contoller和@RequestMapping注解支持类：处理器映射RequestMappingHandlerMapping 和 处理器适配器

新的@ExceptionHandler 注解支持类：ExceptionHandlerExceptionResolver来代替Spring3.0的Annotation

@RequestMapping的"consumes" 和 "produces" 条件支持：``用于支持@RequestBody 和 @ResponseBody，

1``consumes``指定请求的内容是什么类型的内容，即本处理方法消费什么类型的数据，如consumes="application/

2``produces``指定生产什么类型的内容，如produces="application/json"表示JSON类型的内容，Spring的根据

3``以上内容，本章第×××节详述。

URI模板变量增强：``URI模板变量可以直接绑定到@ModelAttribute指定的命令对象、@PathVariable方法参数在视图

@Validated：``JSR-303的javax.validation.Valid一种变体（非JSR-303规范定义的，而是Spring自定义的），

@RequestPart：``提供对“multipart/form-data”请求的全面支持，支持Servlet 3.0文件上传（javax.servlet

Flash 属性 和 RedirectAttribute：``通过FlashMap存储一个请求的输出，当进入另一个请求时作为该请求的输入

Spring Web MVC提供FlashMapManager用于管理FlashMap，默认使用 SessionFlashMapManager，即数据默认存储在session中 。

私塾在线学习网原创内容（http://sishuok.com）

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

3.1、DispatcherServlet作用

DispatcherServlet是前端控制器设计模式的实现，提供Spring Web MVC的集中访问点，而且负责职责的分派，而且与Spring IoC容器无缝集成，从而可以获得Spring的所有好处。 具体请参考第二章的图2-1。

DispatcherServlet主要用作职责调度工作，本身主要用于控制流程，主要职责如下：

1、文件上传解析，如果请求类型是multipart将通过MultipartResolver进行文件上传解析；

2、通过HandlerMapping，将请求映射到处理器（返回一个HandlerExecutionChain，它包括一个处理器、多个HandlerInterceptor拦截器）；

3、通过HandlerAdapter支持多种类型的处理器(HandlerExecutionChain中的处理器)；

4、通过ViewResolver解析逻辑视图名到具体视图实现；

5、本地化解析；

6、渲染具体的视图等；

7、如果执行过程中遇到异常将交给HandlerExceptionResolver来解析。

从以上我们可以看出DispatcherServlet主要负责流程的控制（而且在流程中的每个关键点都是很容易扩展的）。

3.2、DispatcherServlet在web.xml中的配置

```
<servlet>
    <servlet-name>chapter2</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>chapter2</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

load-on-startup：表示启动容器时初始化该Servlet；

url-pattern：表示哪些请求交给Spring Web MVC处理，“/” 是用来定义默认servlet映射的。也可以如“*.html”表示拦截所有以html为扩展名的请求。

该DispatcherServlet默认使用WebApplicationContext作为上下文，Spring默认配置文件为“/WEB-INF/[servlet名字]-servlet.xml”。

DispatcherServlet也可以配置自己的初始化参数，覆盖默认配置：

摘自Spring Reference

参数	描述
contextClass	实现WebApplicationContext接口的类，当前的servlet用它来创建上下文。如果这个参数没有指定， 默认使用XmlWebApplicationContext。
contextConfigLocation	传给上下文实例（由contextClass指定）的字符串，用来指定上下文的位置。这个字符串可以被分成多个字符串（使用逗号作为分隔符） 来支持多个上下文（在多上下文的情况下，如果同一个bean被定义两次，后面一个优先）。
namespace	WebApplicationContext命名空间。默认值是[server-name]-servlet。

因此我们可以通过添加初始化参数

```
<servlet>
    <servlet-name>chapter2</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-servlet-config.xml</param-value>
    </init-param>
</servlet>
```

如果使用如上配置，Spring Web MVC框架将加载“classpath:spring-servlet-config.xml”来进行初始化上下文而不是“/WEB-INF/[servlet名字]-servlet.xml”。

3.3、上下文关系

集成Web环境的通用配置：

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
          classpath:spring-common-config.xml,
          classpath:spring-budget-config.xml
      </param-value>
</context-param>
<listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listen
</listener>
```

如上配置是Spring集成Web环境的通用配置；一般用于加载除Web层的Bean（如DAO、Service等），以便于与其他任何Web框架集成。

contextConfigLocation：表示用于加载Bean的配置文件；

contextClass：表示用于加载Bean的ApplicationContext实现类，默认WebApplicationContext。

创建完毕后会将该上下文放在ServletContext：

servletContext.setAttribute(

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,

this.context);

ContextLoaderListener初始化的上下文和**DispatcherServlet**初始化的上下文关系，如图**3-1**

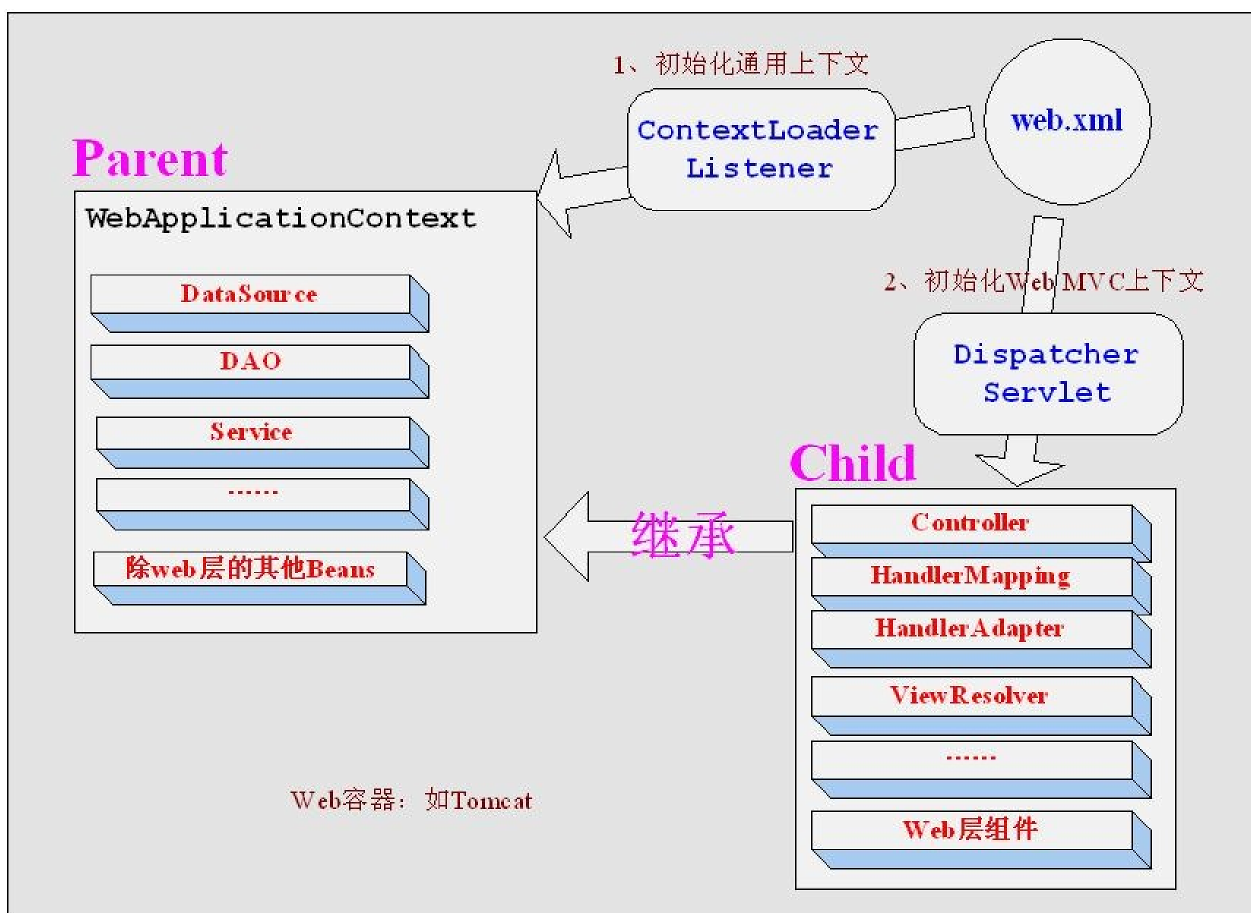


图3-1

从图中可以看出：

ContextLoaderListener初始化的上下文加载的Bean是对于整个应用程序共享的，不管是使用什么表现层技术，一般如DAO层、Service层Bean；

DispatcherServlet初始化的上下文加载的Bean是只对Spring Web MVC有效的Bean，如Controller、HandlerMapping、HandlerAdapter等等，该初始化上下文应该只加载Web相关组件。

3.4、DispatcherServlet初始化顺序

继承体系结构如下所示：



1、**HttpServletBean**继承**HttpServlet**，因此在Web容器启动时将调用它的init方法，该初始化方法的主要作用

：：：将Servlet初始化参数（init-param）设置到该组件上（如contextAttribute、contextClass、namespace、contextConfigLocation），通过BeanWrapper简化设值过程，方便后续使用；

：：：提供给子类初始化扩展点，initServletBean()，该方法由FrameworkServlet覆盖。

```
public abstract class HttpServletBean extends HttpServlet implements EnvironmentAware{
@Override
    public final void init() throws ServletException {
       //省略部分代码
       //1、如下代码的作用是将Servlet初始化参数设置到该组件上
//如contextAttribute、contextClass、namespace、contextConfigLocation；
       try {
           PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.
           BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
           ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletCon
           bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, thi
           initBeanWrapper(bw);
           bw.setPropertyValues(pvs, true);
       }
       catch (BeansException ex) {
           //…………省略其他代码
       }
       //2、提供给子类初始化的扩展点，该方法由FrameworkServlet覆盖
       initServletBean();
       if (logger.isDebugEnabled()) {
           logger.debug("Servlet '" + getServletName() + "' configured successfully");
       }
    }
    //…………省略其他代码
}
```

2、**FrameworkServlet**继承**HttpServletBean**，通过initServletBean()进行Web上下文初始化，该方法主要覆盖一下两件事情：

初始化web上下文；

提供给子类初始化扩展点；

```
public abstract class FrameworkServlet extends HttpServletBean {
@Override
    protected final void initServletBean() throws ServletException {
        //省略部分代码
        try {
            //1、初始化Web上下文
            this.webApplicationContext = initWebApplicationContext();
            //2、提供给子类初始化的扩展点
            initFrameworkServlet();
        }
        //省略部分代码
    }
}
```

```
protected WebApplicationContext initWebApplicationContext() {
        //ROOT上下文（ContextLoaderListener加载的）
        WebApplicationContext rootContext =
               WebApplicationContextUtils.getWebApplicationContext(getServletContext());
        WebApplicationContext wac = null;
        if (this.webApplicationContext != null) {
            // 1、在创建该Servlet注入的上下文
            wac = this.webApplicationContext;
            if (wac instanceof ConfigurableWebApplicationContext) {
               ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext
               if (!cwac.isActive()) {
                   if (cwac.getParent() == null) {
                       cwac.setParent(rootContext);
                   }
                   configureAndRefreshWebApplicationContext(cwac);
               }
            }
        }
        if (wac == null) {
            //2、查找已经绑定的上下文
            wac = findWebApplicationContext();
        }
        if (wac == null) {
            //3、如果没有找到相应的上下文，并指定父亲为ContextLoaderListener
            wac = createWebApplicationContext(rootContext);
        }
        if (!this.refreshEventReceived) {
            //4、刷新上下文（执行一些初始化）
            onRefresh(wac);
        }
        if (this.publishContext) {
            // Publish the context as a servlet context attribute.
            String attrName = getServletContextAttributeName();
            getServletContext().setAttribute(attrName, wac);
            //省略部分代码
        }
        return wac;
    }
```

从initWebApplicationContext（）方法可以看出，基本上如果ContextLoaderListener加载了上下文将作为根上下文（DispatcherServlet的父容器）。

最后调用了onRefresh()方法执行容器的一些初始化，这个方法由子类实现，来进行扩展。

3、DispatcherServlet继承**FrameworkServlet**，并实现了onRefresh()方法提供一些前端控制器相关的配置：

```
public class DispatcherServlet extends FrameworkServlet {
     //实现子类的onRefresh()方法，该方法委托为initStrategies()方法。
    @Override
    protected void onRefresh(ApplicationContext context) {
       initStrategies(context);
    }

    //初始化默认的Spring Web MVC框架使用的策略（如HandlerMapping）
    protected void initStrategies(ApplicationContext context) {
       initMultipartResolver(context);
       initLocaleResolver(context);
       initThemeResolver(context);
       initHandlerMappings(context);
       initHandlerAdapters(context);
       initHandlerExceptionResolvers(context);
       initRequestToViewNameTranslator(context);
       initViewResolvers(context);
       initFlashMapManager(context);
    }
}
```

从如上代码可以看出，DispatcherServlet启动时会进行我们需要的Web层Bean的配置，如HandlerMapping、HandlerAdapter等，而且如果我们没有配置，还会给我们提供默认的配置。

从如上代码我们可以看出，整个DispatcherServlet初始化的过程和做了些什么事情，具体主要做了如下两件事情：

1、初始化Spring Web MVC使用的Web上下文，并且可能指定父容器为（ContextLoaderListener加载了根上下文）；

2、初始化DispatcherServlet使用的策略，如HandlerMapping、HandlerAdapter等。

服务器启动时的日志分析（此处加上了ContextLoaderListener从而启动ROOT上下文容器）：

信息: Initializing Spring root WebApplicationContext **//由ContextLoaderListener启动ROOT上下文**

2012-03-12 13:33:55 [main] INFO **org.springframework.web.context.ContextLoader** - Root WebApplicationContext: initialization started

2012-03-12 13:33:55 [main] INFO org.springframework.web.context.support.XmlWebApplicationContext - Refreshing Root WebApplicationContext: startup date [Mon Mar 12 13:33:55 CST 2012]; root of context hierarchy

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader - Loading bean definitions

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loaded 0 bean definitions from location pattern **[/WEB-INF/ContextLoaderListener.xml]**

2012-03-12 13:33:55 [main] DEBUG org.springframework.web.context.support.XmlWebApplicationContext - Bean factory for Root WebApplicationContext: org.springframework.beans.factory.support.DefaultListableBeanFactory@1c05ffd: defining beans []; root of factory hierarchy

2012-03-12 13:33:55 [main] DEBUG org.springframework.web.context.support.XmlWebApplicationContext - Bean factory for Root WebApplicationContext:

2012-03-12 13:33:55 [main] DEBUG org.springframework.web.context.ContextLoader - Published root WebApplicationContext as ServletContext attribute with name [org.springframework.web.context.WebApplicationContext.ROOT] //将ROOT上下文绑定到ServletContext

2012-03-12 13:33:55 [main] INFO org.springframework.web.context.ContextLoader - Root WebApplicationContext: initialization completed in 438 ms **//到此ROOT上下文启动完毕**

2012-03-12 13:33:55 [main] DEBUG org.springframework.web.servlet.DispatcherServlet - Initializing servlet 'chapter2'

信息: Initializing Spring FrameworkServlet 'chapter2' **//开始初始化FrameworkServlet对应的Web上下文**

2012-03-12 13:33:55 [main] INFO org.springframework.web.servlet.DispatcherServlet - FrameworkServlet 'chapter2': initialization started

2012-03-12 13:33:55 [main] DEBUG org.springframework.web.servlet.DispatcherServlet - Servlet with name 'chapter2' will try to **create custom WebApplicationContext context** of class 'org.springframework.web.context.support.XmlWebApplicationContext', **using parent context [Root WebApplicationContext**: startup date [Mon Mar 12 13:33:55 CST 2012]; root of context hierarchy]

//此处使用Root WebApplicationContext作为父容器。

2012-03-12 13:33:55 [main] INFO org.springframework.web.context.support.XmlWebApplicationContext - Refreshing WebApplicationContext for namespace 'chapter2-servlet': startup date [Mon Mar 12 13:33:55 CST 2012]; parent: Root WebApplicationContext

2012-03-12 13:33:55 [main] INFO org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from ServletContext resource **[/WEB-INF/chapter2-servlet.xml]**

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader - Loading bean definitions

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.BeanDefinitionParserDelegate - Neither XML 'id' nor 'name' specified - using generated bean name**[org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping#0] //**我们配置的**HandlerMapping**

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.BeanDefinitionParserDelegate - Neither XML 'id' nor 'name' specified - using generated bean name**[org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter#0] //**我们配置的**HandlerAdapter**

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.BeanDefinitionParserDelegate - Neither XML 'id' nor 'name' specified - using generated bean name [org.springframework.web.servlet.view.InternalResourceViewResolver#0] **//**我们配置的**ViewResolver**

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.BeanDefinitionParserDelegate - No XML 'id' specified - using '/hello' as bean name and [] as aliases

//我们的处理器（**HelloWorldController**）

2012-03-12 13:33:55 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loaded 4 bean definitions from location pattern [/WEB-INF/chapter2-servlet.xml]

2012-03-12 13:33:55 [main] DEBUG org.springframework.web.context.support.XmlWebApplicationContext - Bean factory for WebApplicationContext for namespace 'chapter2-servlet': org.springframework.beans.factory.support.DefaultListableBeanFactory@1372656: defining beans [org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping#0,org.springframe

work.web.servlet.mvc.SimpleControllerHandlerAdapter#0,org.springframework.web.servlet.view.InternalResourceViewResolver#0,/hello]; parent: org.springframework.beans.factory.support.DefaultListableBeanFactory@1c05ffd

//到此容器注册的**Bean**初始化完毕

2012-03-12 13:33:56 [main] DEBUG org.springframework.web.servlet.DispatcherServlet - Unable to locate MultipartResolver with name 'multipartResolver': no multipart request handling provided

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating instance of bean 'org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver'

//默认的**LocaleResolver**注册

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating instance of bean 'org.springframework.web.servlet.theme.FixedThemeResolver'

//默认的**ThemeResolver**注册

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Returning cached instance of singleton bean 'org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping#0'

//发现我们定义的**HandlerMapping** 不再使用默认的**HandlerMapping**。

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Returning cached instance of singleton bean 'org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter#0'

//发现我们定义的**HandlerAdapter** 不再使用默认的**HandlerAdapter**。

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating instance of bean 'org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver'

//异常处理解析器**ExceptionResolver**

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating instance of bean 'org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver'

2012-03-12 13:33:56 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Returning cached instance of singleton bean 'org.springframework.web.servlet.view.InternalResourceViewResolver#0'

2012-03-12 13:33:56 [main] DEBUG org.springframework.web.servlet.DispatcherServlet - Published WebApplicationContext of servlet 'chapter2' as ServletContext attribute with name [org.springframework.web.servlet.FrameworkServlet.CONTEXT.chapter2]

//绑定FrameworkServlet初始化的Web上下文到ServletContext

2012-03-12 13:33:56 [main] INFO org.springframework.web.servlet.DispatcherServlet - FrameworkServlet 'chapter2': initialization completed in 297 ms

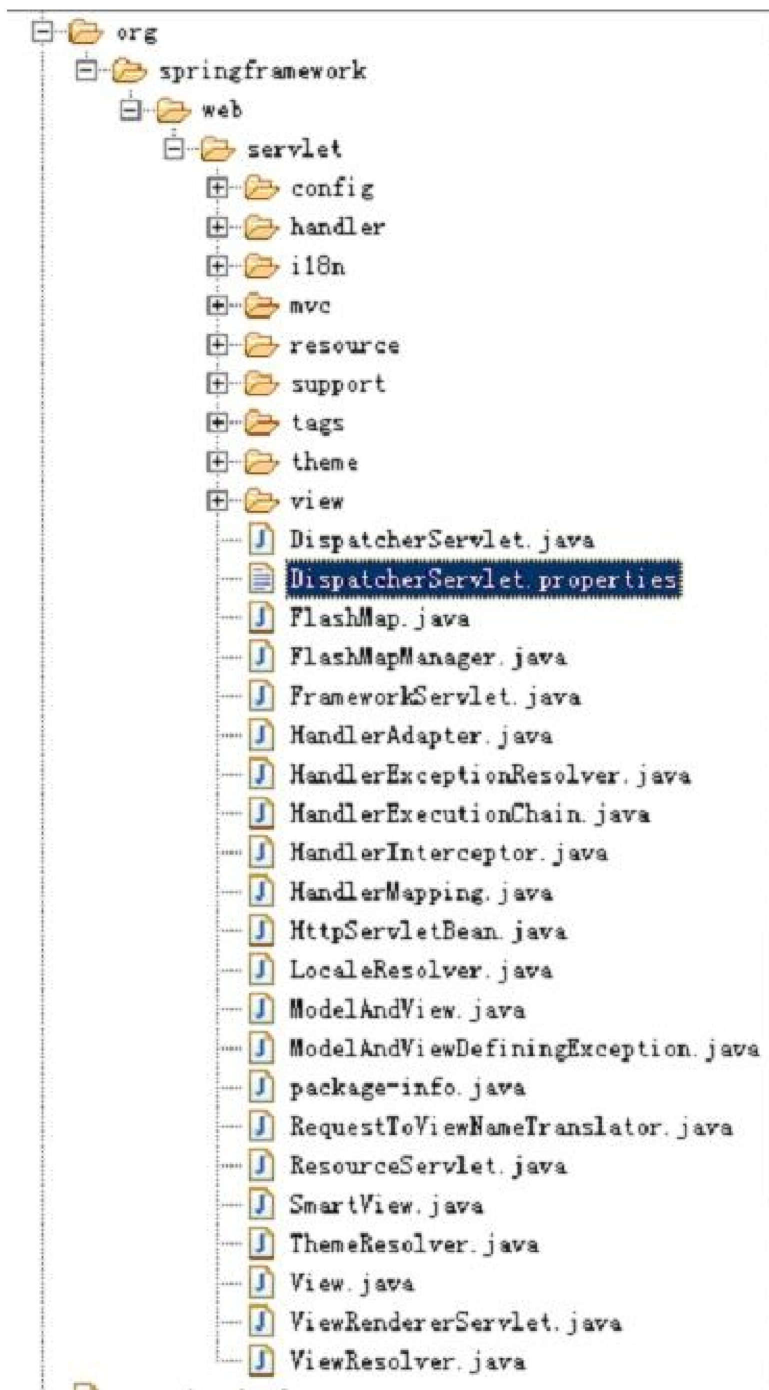
2012-03-12 13:33:56 [main] DEBUG org.springframework.web.servlet.DispatcherServlet - Servlet 'chapter2' configured successfully

//到此完整流程结束

从如上日志我们也可以看出，DispatcherServlet会进行一些默认的配置。接下来我们看一下默认配置吧。

3.5、DispatcherServlet默认配置

DispatcherServlet的默认配置在DispatcherServlet.properties（和DispatcherServlet类在一个包下）中，而且是当Spring配置文件中没有指定配置时使用的默认策略：



org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\

org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\

org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager

从如上配置可以看出DispatcherServlet在启动时会自动注册这些特殊的Bean，无需我们注册，如果我们注册了，默认的将不会注册。

因此如第二章的BeanNameUrlHandlerMapping、SimpleControllerHandlerAdapter是不需要注册的，DispatcherServlet默认会注册这两个Bean。

从DispatcherServlet.properties可以看出有许多特殊的Bean，那接下来我们就看看Spring Web MVC主要有哪些特殊的Bean。

3.6、DispatcherServlet中使用的特殊的Bean

DispatcherServlet默认使用WebApplicationContext作为上下文，因此我们来看一下该上下文中有哪些特殊的Bean：

1、Controller：处理器/页面控制器，做的是MVC中的C的事情，但控制逻辑转移到前端控制器了，用于对请求进行处理；

2、HandlerMapping：请求到处理器的映射，如果映射成功返回一个HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象；如BeanNameUrlHandlerMapping将URL与Bean名字映射，映射成功的Bean就是此处的处理器；

3、HandlerAdapter：HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；如SimpleControllerHandlerAdapter将对实现了Controller接口的Bean进行适配，并且掉处理器的handleRequest方法进行功能处理；

4、ViewResolver：ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；如InternalResourceViewResolver将逻辑视图名映射为jsp视图；

5、LocalResover：本地化解析，因为Spring支持国际化，因此LocalResover解析客户端的Locale信息从而方便进行国际化；

6、ThemeResovler：主题解析，通过它来实现一个页面多套风格，即常见的类似于软件皮肤效果；

7、MultipartResolver：文件上传解析，用于支持文件上传；

8、HandlerExceptionResolver：处理器异常解析，可以将异常映射到相应的统一错误界面，从而显示用户友好的界面（而不是给用户看到具体的错误信息）；

9、RequestToViewNameTranslator：当处理器没有返回逻辑视图名等相关信息时，自动将请求URL映射为逻辑视图名；

10、FlashMapManager：用于管理FlashMap的策略接口，FlashMap用于存储一个请求的输出，当进入另一个请求时作为该请求的输入，通常用于重定向场景，后边会细述。

到此DispatcherServlet我们已经了解了，接下来我们就需要把上边提到的特殊Bean挨个击破，那首先从控制器开始吧。

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

4.1、Controller简介

Controller控制器，是MVC中的部分C，为什么是部分呢？因为此处的控制器主要负责功能处理部分：

1、收集、验证请求参数并绑定到命令对象；

2、将命令对象交给业务对象，由业务对象处理并返回模型数据；

3、返回ModelAndView（Model部分是业务对象返回的模型数据，视图部分为逻辑视图名）。

还记得DispatcherServlet吗？主要负责整体的控制流程的调度部分：

1、负责将请求委托给控制器进行处理；

2、根据控制器返回的逻辑视图名选择具体的视图进行渲染（并把模型数据传入）。

因此MVC中完整的C（包含控制逻辑+功能处理）由（DispatcherServlet + Controller）组成。

因此此处的控制器是Web MVC中部分，也可以称为页面控制器、动作、处理器。

Spring Web MVC支持多种类型的控制器，比如实现Controller接口，从Spring2.5开始支持注解方式的控制器（如@Controller、@RequestMapping、@RequestParam、@ModelAttribute等），我们也可以自己实现相应的控制器（只需要定义相应的HandlerMapping和HandlerAdapter即可）。

因为考虑到还有部分公司使用继承Controller接口实现方式，因此我们也学习一下，虽然已经不推荐使用了。

对于注解方式的控制器，后边会详细讲，在此我们先学习Spring2.5以前的Controller接口实现方式。

首先我们将项目springmvc-chapter2复制一份改为项目springmvc-chapter4，本章示例将放置在springmvc-chapter4中。

大家需要将项目springmvc-chapter4/ .settings/ org.eclipse.wst.common.component下的chapter2改为chapter4，否则上下文还是“springmvc-chapter2”。以后的每一个章节都需要这么做。

4.2、Controller接口

```
package org.springframework.web.servlet.mvc;
public interface Controller {
       ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse respons
}
```

这是控制器接口，此处只有一个方法handleRequest，用于进行请求的功能处理，处理完请求后返回ModelAndView（Model模型数据部分 和 View视图部分）。

还记得第二章的HelloWorld吗？我们的HelloWorldController实现Controller接口，Spring默认提供了一些Controller接口的实现以方便我们使用，具体继承体系如图4-1：

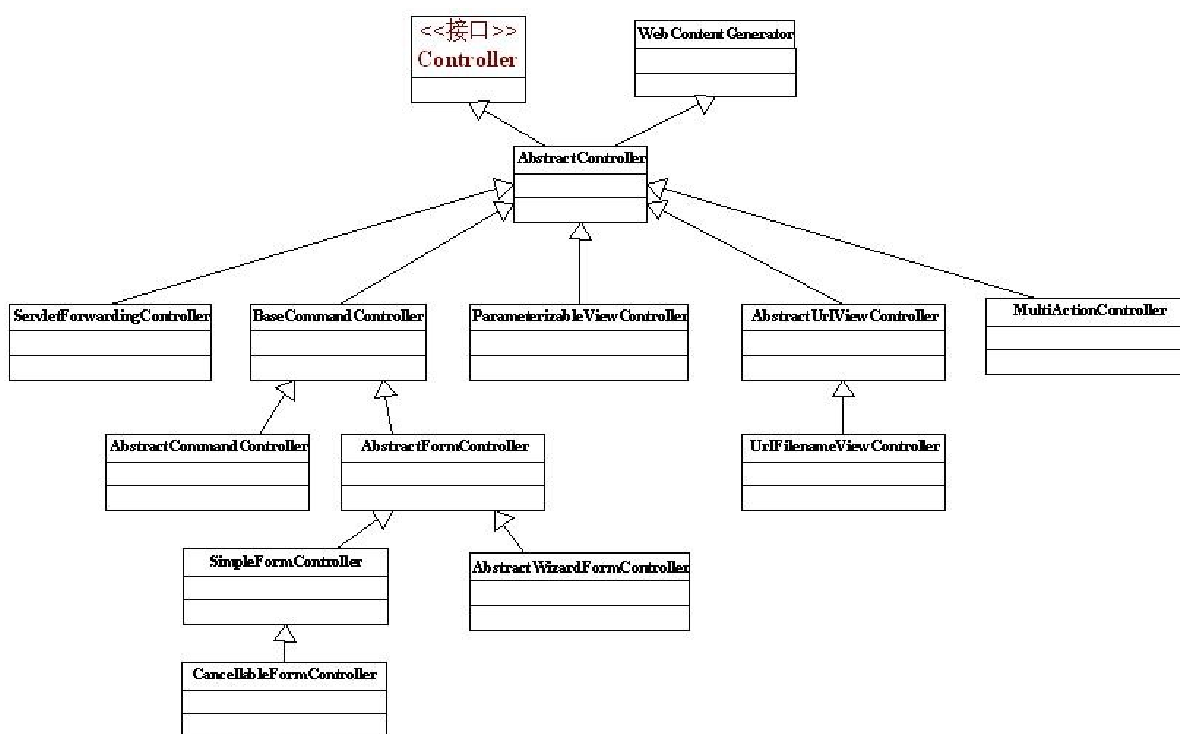


图4-1

4.3、WebContentGenerator

用于提供如浏览器缓存控制、是否必须有session开启、支持的请求方法类型（GET、POST等）等，该类主要有如下属性：

Set<String> supportedMethods：设置支持的请求方法类型，默认支持"GET"、"POST"、"HEAD"，如果我们想支持"PUT"，则可以加入该集合"PUT"。

boolean requireSession = false：是否当前请求必须有session，如果此属性为true，但当前请求没有打开session将抛出HttpSessionRequiredException异常；

boolean useExpiresHeader = true：是否使用HTTP1.0协议过期响应头：如果true则会在响应头添加："Expires："；需要配合cacheSeconds使用；

boolean useCacheControlHeader = true：是否使用HTTP1.1协议的缓存控制响应头，如果true则会在响应头添加；需要配合cacheSeconds使用；

boolean useCacheControlNoStore = true：是否使用HTTP 1.1协议的缓存控制响应头，如果true则会在响应头添加；需要配合cacheSeconds使用；

private int cacheSeconds = -1：缓存过期时间，正数表示需要缓存，负数表示不做任何事情（也就是说保留上次的缓存设置），

1、cacheSeconds **=**0时，则将设置如下响应头数据：

Pragma：no-cache // HTTP 1.0的不缓存响应头

Expires：1L // useExpiresHeader=true时，HTTP 1.0

Cache-Control：**no-cache** // useCacheControlHeader=true时，HTTP 1.1

Cache-Control：**no-store** // useCacheControlNoStore=true时，该设置是防止Firefox缓存

2、cacheSeconds>0时，则将设置如下响应头数据：

Expires：System.currentTimeMillis() + **cacheSeconds** * 1000L // useExpiresHeader=true时，HTTP 1.0

Cache-Control：**max-age=cacheSeconds** // useCacheControlHeader=true时，HTTP 1.1

3、cacheSeconds<0时，则什么都不设置，即保留上次的缓存设置。

此处简单说一下以上响应头的作用，缓存控制已超出本书内容：

HTTP1.0缓存控制响应头

Pragma：no-cache：表示防止客户端缓存，需要强制从服务器获取最新的数据；

Expires：HTTP1.0响应头，本地副本缓存过期时间，如果客户端发现缓存文件没有过期则不发送请求，HTTP的日期时间必须是格林威治时间（GMT）， 如“Expires:Wed, 14 Mar 2012 09:38:32 GMT”；

HTTP1.1缓存控制响应头

Cache-Control：no-cache 强制客户端每次请求获取服务器的最新版本，不经过本地缓存的副本验证；

Cache-Control：no-store强制客户端不保存请求的副本，该设置是防止Firefox缓存

Cache-Control：max-age=[秒] 客户端副本缓存的最长时间，类似于HTTP1.0的Expires，只是此处是基于请求的相对时间间隔来计算，而非绝对时间。

还有相关缓存控制机制如Last-Modified（最后修改时间验证，客户端的上一次请求时间 在 服务器的最后修改时间 之后，说明服务器数据没有发生变化 返回304状态码）、ETag（没有变化时不重新下载数据，返回304）。

该抽象类默认被AbstractController和WebContentInterceptor继承。

4.4、AbstractController

该抽象类实现了Controller，并继承了WebContentGenerator（具有该类的特性，具体请看4.3），该类有如下属性：

boolean synchronizeOnSession = false：表示该控制器是否在执行时同步session，从而保证该会话的用户串行访问该控制器。

```
public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse respons
              //委托给WebContentGenerator进行缓存控制
              checkAndPrepare(request, response, this instanceof LastModified);
              //当前会话是否应串行化访问.
              if (this.synchronizeOnSession) {
                     HttpSession session = request.getSession(false);
                     if (session != null) {
                            Object mutex = WebUtils.getSessionMutex(session);
                            synchronized (mutex) {
                                   return handleRequestInternal(request, response);
                            }
                     }
              }
              return handleRequestInternal(request, response);
}
```

可以看出AbstractController实现了一些特殊功能，如继承了WebContentGenerator缓存控制功能，并提供了可选的会话的串行化访问功能。而且提供了handleRequestInternal方法，因此我们应该在具体的控制器类中实现handleRequestInternal方法，而不再是handleRequest。

AbstractController使用方法：

首先让我们使用AbstractController来重写第二章的HelloWorldController：

```
public class HelloWorldController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo
        //1、收集参数
        //2、绑定参数到命令对象
        //3、调用业务对象
        //4、选择下一个页面
        ModelAndView mv = new ModelAndView();
        //添加模型数据 可以是任意的POJO对象
        mv.addObject("message", "Hello World!");
        //设置逻辑视图名，视图解析器会根据该名字解析到具体的视图页面
        mv.setViewName("hello");
        return mv;
    }
}
```

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/hello" class="cn.javass.chapter4.web.controller.HelloWorldController"/>
```

从如上代码我们可以看出：

1、继承AbstractController

2、实现handleRequestInternal方法即可。

直接通过**response**写响应

如果我们想直接在控制器通过response写出响应呢，以下代码帮我们阐述：

```
public class HelloWorldWithoutReturnModelAndViewController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo

        resp.getWriter().write("Hello World!!");
        //如果想直接在该处理器/控制器写响应 可以通过返回null告诉DispatcherServlet自己已经写出响应了，
        return null;
    }
}
```

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/helloWithoutReturnModelAndView" class="cn.javass.chapter4.web.controller.Hel
```

从如上代码可以看出如果想直接在控制器写出响应，只需要通过response写出，并返回null即可。

强制请求方法类型：

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/helloWithPOST" class="cn.javass.chapter4.web.controller.HelloWorldController
        <property name="supportedMethods" value="POST"></property>
</bean>
```

以上配置表示只支持POST请求，如果是GET请求客户端将收到“HTTP Status 405 - Request method 'GET' not supported”。

比如注册/登录可能只允许POST请求。

当前请求的**session**前置条件检查，如果当前请求无**session**将抛出**HttpSessionRequiredException**异常：

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/helloRequireSession"
class="cn.javass.chapter4.web.controller.HelloWorldController">
        <property name="requireSession" value="true"/>
</bean>
```

```
在进入该控制器时，一定要有session存在，否则抛出HttpSessionRequiredException异常。
```

Session同步：

即同一会话只能串行访问该控制器。

客户端端缓存控制：

1、缓存**5**秒，**cacheSeconds=5**

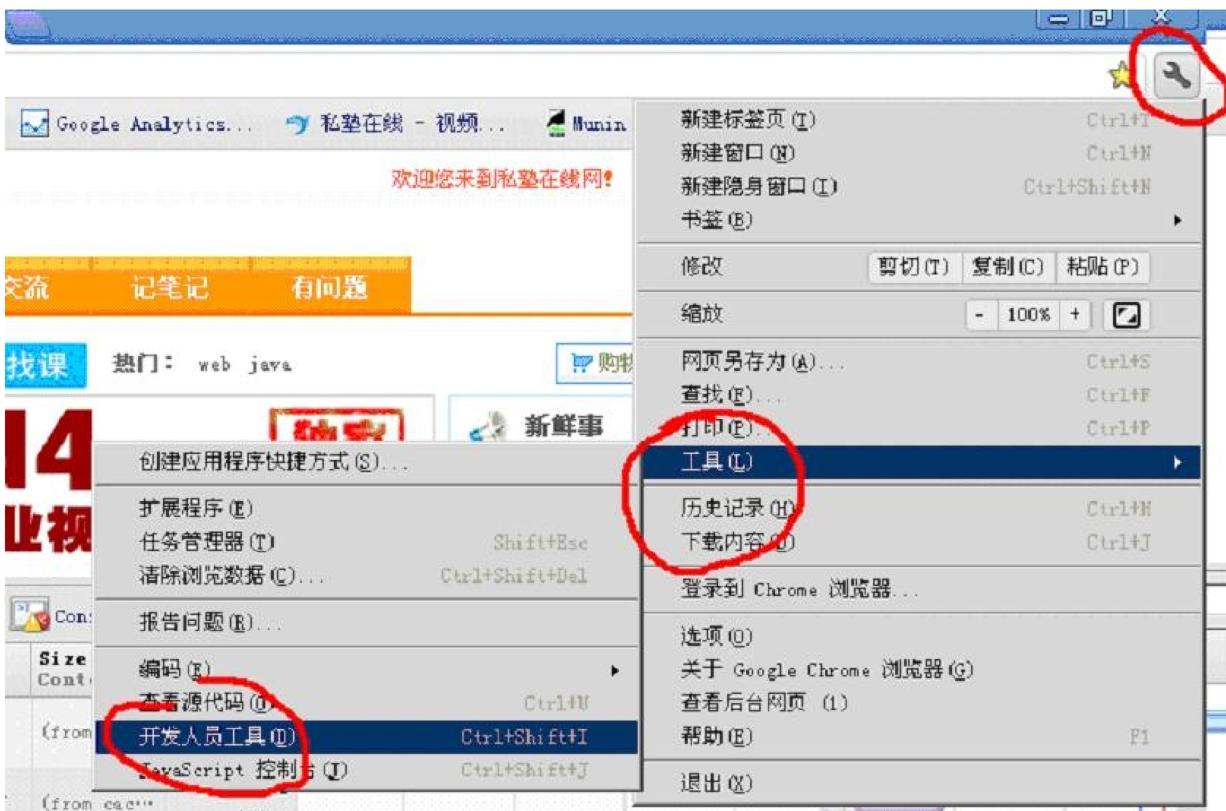
```
package cn.javass.chapter4.web.controller;
//省略import
public class HelloWorldCacheController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo

        //点击后再次请求当前页面
        resp.getWriter().write("<a href=''>this</a>");
        return null;
    }
}
```

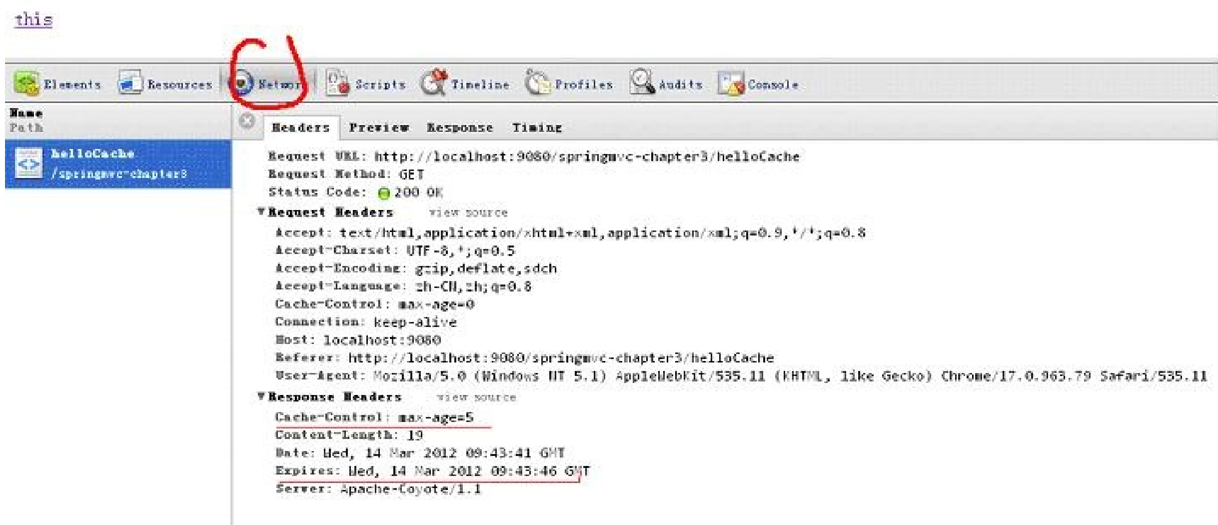
```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/helloCache"
class="cn.javass.chapter4.web.controller.HelloWorldCacheController">
<property name="cacheSeconds" value="5"/>
</bean>
```

如上配置表示告诉浏览器缓存**5**秒钟：

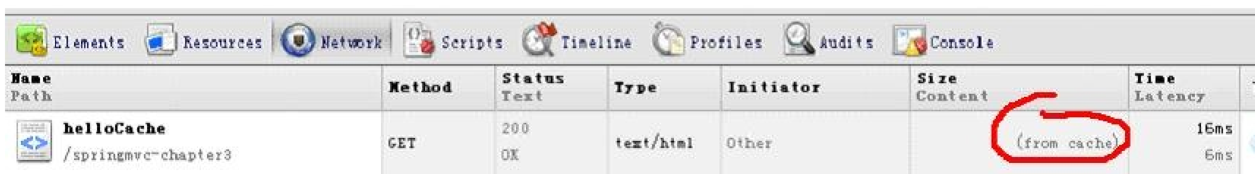
开启chrome浏览器调试工具：



服务器返回的响应头如下所示：



添加了“Expires:Wed, 14 Mar 2012 09:38:32 GMT” 和“Cache-Control:max-age=5” 表示允许客户端缓存5秒，当你点“this”链接时，会发现如下：



而且服务器也没有收到请求，当过了5秒后，你再点“this”链接会发现又重新请求服务器下载新数据。

注：下面提到一些关于缓存控制的一些特殊情况：

1、对于一般的页面跳转（如超链接点击跳转、通过js调用window.open打开新页面都是会使用浏览器缓存的，在未过期情况下会直接使用浏览器缓存的副本，在未过期情况下一次请求也不发送）；

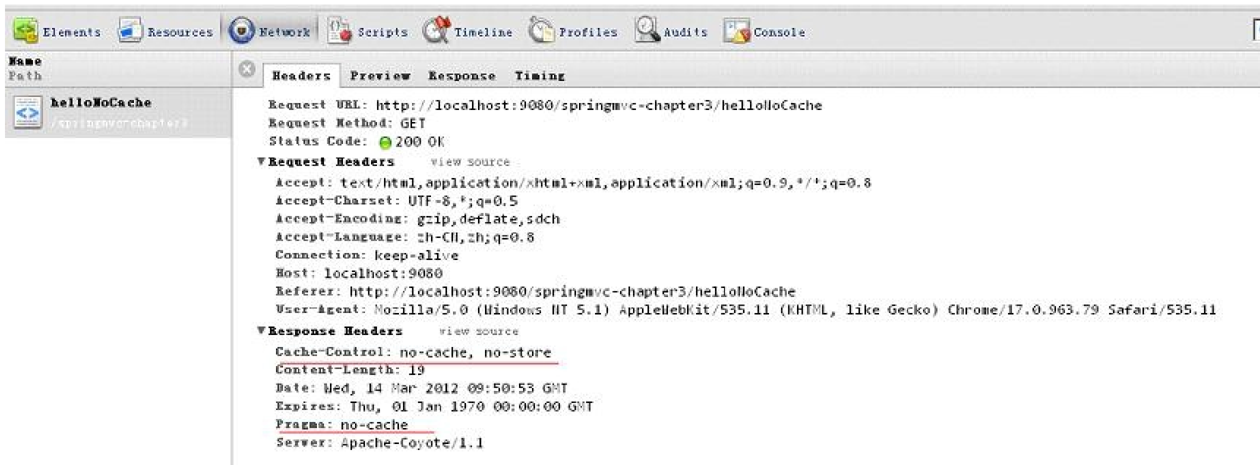
2、对于刷新页面（如按F5键刷新），会再次发送一次请求到服务器的；

2、不缓存，cacheSeconds=0

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/helloNoCache"
class="cn.javass.chapter4.web.controller.HelloWorldCacheController">
<property name="cacheSeconds" value="0"/>
</bean>
```

以上配置会要求浏览器每次都去请求服务器下载最新的数据：

this



3、cacheSeconds<0，将不添加任何数据

响应头什么缓存控制信息也不加。

4、Last-Modified缓存机制

（1、在客户端第一次输入url时，服务器端会返回内容和状态码200表示请求成功并返回了内容；同时会添加一个"Last-Modified"的响应头表示此文件在服务器上的最后更新时间，如"Last-Modified:Wed, 14 Mar 2012 10:22:42 GMT"表示最后更新时间为（2012-03-14 10：22）；

（2、客户端第二次请求此URL时，客户端会向服务器发送请求头 "If-Modified-Since"，询问服务器该时间之后当前请求内容是否有被修改过，如"If-Modified-Since: Wed, 14 Mar 2012 10:22:42 GMT"，如果服务器端的内容没有变化，则自动返回 HTTP 304状态码（只要响应头，内容为空，这样就节省了网络带宽）。

客户端强制缓存过期：

（1、可以按ctrl+F5强制刷新（会添加请求头 HTTP1.0 Pragma:no-cache和 HTTP1.1 Cache-Control:no-cache、If-Modified-Since请求头被删除）表示强制获取服务器内容，不缓存。

（2、在请求的url后边加上时间戳来重新获取内容，加上时间戳后浏览器就认为不是同一份内容：

http://sishuok.com/?2343243243 和 http://sishuok.com/?34334344 是两次不同的请求。

Spring也提供了**Last-Modified**机制的支持，只需要实现**LastModified**接口，如下所示：

```
package cn.javass.chapter4.web.controller;
public class HelloWorldLastModifiedCacheController extends AbstractController implements
    private long lastModified;
    protected ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletRespo
        //点击后再次请求当前页面
        resp.getWriter().write("<a href=''>this</a>");
        return null;
    }
    public long getLastModified(HttpServletRequest request) {
        if(lastModified == 0L) {
            //TODO 此处更新的条件：如果内容有更新，应该重新返回内容最新修改的时间戳
            lastModified = System.currentTimeMillis();
        }
        return lastModified;
    }
}
```

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/helloLastModified"
class="cn.javass.chapter4.web.controller.HelloWorldLastModifiedCacheController"/>
```

HelloWorldLastModifiedCacheController只需要实现LastModified接口的getLastModified方法，保证当内容发生改变时返回最新的修改时间即可。

分析：

（1、发送请求到服务器，如（http://localhost:9080/springmvc-chapter4/helloLastModified），则服务器返回的响应为：



（2、再次按F5刷新客户端，返回状态码304表示服务器没有更新过：



（3、重启服务器，再次刷新，会看到200状态码（因为服务器的lastModified时间变了）。

Spring判断是否过期，通过如下代码，即请求的**“If-Modified-Since”** 大于等于当前的**getLastModified**方法的时间戳，则认为没有修改：

this.notModified = (ifModifiedSince >= (lastModifiedTimestamp / 1000 * 1000));

5、**ETag**（实体标记）缓存机制

（1：浏览器第一次请求，服务器在响应时给请求URL标记，并在HTTP响应头中将其传送到客户端，类似服务器端返回的格式：“ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"”

（2：浏览器第二次请求，客户端的查询更新格式是这样的：“If-None-Match:"0f8b0c86fe2c0c7a67791e53d660208e3"”，如果ETag没改变，表示内容没有发生改变，则返回状态304。

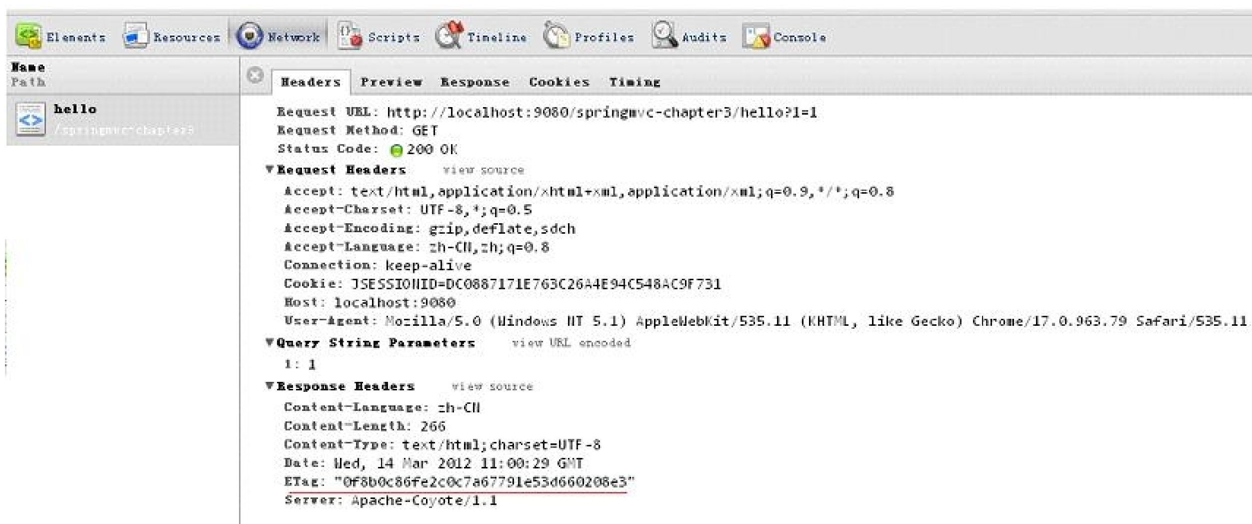
Spring也提供了对**ETag**的支持，具体需要在**web.xml**中配置如下代码：

```
<filter>
   <filter-name>etagFilter</filter-name>
   <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
</filter>
<filter-mapping>
   <filter-name>etagFilter</filter-name>
   <servlet-name>chapter4</servlet-name>
</filter-mapping>
```

此过滤器只过滤到我们DispatcherServlet的请求。

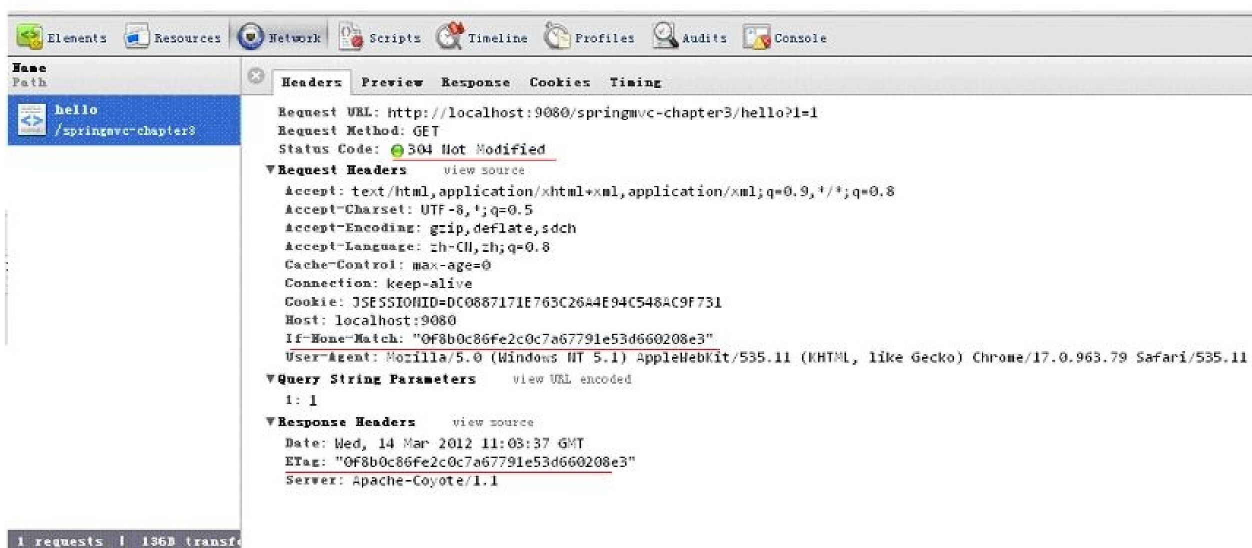
分析：

1）：发送请求到服务器：“http://localhost:9080/springmvc-chapter4/hello”，服务器返回的响应头中添加了（ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"）：



2）：浏览器再次发送请求到服务器（按F5刷新），请求头中添加了“If-None-Match:

"0f8b0c86fe2c0c7a67791e53d660208e3"”，响应返回304代码，表示服务器没有修改，并且响应头再次添加了“ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"”（每次都需要计算）：



那服务器端是如何计算**ETag**的呢？

```
protected String generateETagHeaderValue(byte[] bytes) {
              StringBuilder builder = new StringBuilder("\"0");
              DigestUtils.appendMd5DigestAsHex(bytes, builder);
              builder.append('"');
              return builder.toString();
}
```

bytes是response要写回到客户端的响应体（即响应的内容数据），是通过MD5算法计算的内容的摘要信息。也就是说如果服务器内容不发生改变，则ETag每次都是一样的，即服务器端的内容没有发生改变。

此处只列举了部分缓存控制，详细介绍超出了本书的范围，强烈推荐：**http://www.mnot.net/cache_docs/**（中文版**http://www.chedong.com/tech/cache_docs.html**）详细了解**HTTP**缓存控制及为什么要

缓存。

缓存的目的是减少相应延迟 和 减少网络带宽消耗，比如**css**、**js**、图片这类静态资源应该进行缓存。

实际项目一般使用反向代理服务器（如**nginx**、**apache**等）进行缓存。

[

私塾在线学习网](http://sishuok.com/)原创内容（http://sishuok.com）

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/0/5234.html】

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

4.5、ServletForwardingController

将接收到的请求转发到一个命名的servlet，具体示例如下：

```
package cn.javass.chapter4.web.servlet;
public class ForwardingServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

        resp.getWriter().write("Controller forward to Servlet");

    }
}
```

```
    <servlet>
        <servlet-name>forwarding</servlet-name>
        <servlet-class>cn.javass.chapter4.web.servlet.ForwardingServlet</servlet-class>
    </servlet>
```

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/forwardToServlet"
class="org.springframework.web.servlet.mvc.ServletForwardingController">
        <property name="servletName" value="forwarding"></property>
</bean>
```

当我们请求/forwardToServlet时，会被转发到名字为“forwarding”的servlet处理，该sevlet的servlet-mapping标签配置是可选的。

4.6、BaseCommandController

命令控制器通用基类，提供了以下功能支持：

1、数据绑定：请求参数绑定到一个command object（命令对象，非GoF里的命令设计模式），这里的命令对象是指绑定请求参数的任何POJO对象；

commandClass：表示命令对象实现类，如UserModel；

commandName：表示放入请求的命令对象名字（默认command），request.setAttribute(commandName, commandObject);

2、验证功能：提供Validator注册功能，注册的验证器会验证命令对象属性数据是否合法；

validators：通过该属性注入验证器，验证器用来验证命令对象属性是否合法；

该抽象类没有没有提供流程功能，只是提供了一些公共的功能，实际使用时需要使用它的子类。

4.7、AbstractCommandController

命令控制器之一，可以实现该控制器来创建命令控制器，该控制器能把自动封装请求参数到一个命令对象，而且提供了验证功能。

1、创建命令类（就是普通的JavaBean类/POJO）

```
package cn.javass.chapter4.model;
public class UserModel {
    private String username;
    private String password;
        //省略setter/getter
}
```

2、实现控制器

```
package cn.javass.chapter4.web.controller;
//省略import
public class MyAbstractCommandController extends AbstractCommandController {
    public MyAbstractCommandController() {
        //设置命令对象实现类
        setCommandClass(UserModel.class);
    }
    @Override
    protected ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Objec
        //将命令对象转换为实际类型
        UserModel user = (UserModel) command;
        ModelAndView mv = new ModelAndView();
        mv.setViewName("abstractCommand");
        mv.addObject("user", user);
        return mv;
    }
}
```

```
<!— 在chapter4-servlet.xml配置处理器 -->
<bean name="/abstractCommand"
class="cn.javass.chapter4.web.controller.MyAbstractCommandController">
        <!-- 也可以通过依赖注入 注入命令实现类 -->
        <!-- property name="commandClass" value="cn.javass.chapter4.model.UserModel"/-->
</bean>
```

```
<!— WEB-INF/jsp/abstractCommand.jsp视图下的主要内容 -->

${user.username }-${user.password }
```

当我们在浏览器中输入“http://localhost:9080/springmvc-chapter4/abstractCommand?username=123&password=123”，会自动将请求参数username和password绑定到命令对象；绑定时按照JavaBean命名规范绑定；

```
Host: localhost:9080
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.9
▼Query String Parameters    view URL encoded
  username: 123                                command.setUsername("123");
  password: 123                                command.setPassword("123");
▼Response Headers    view source
  Content-Language: zh-CN
```

4.8、AbstractFormController

用于支持带步骤的表单提交的命令控制器基类，使用该控制器可以完成：

1、定义表单处理（表单的渲染），并从控制器获取命令对象构建表单；

2、提交表单处理，当用户提交表单内容后，AbstractFormController可以将用户请求的数据绑定到命令对象，并可以验证表单内容、对命令对象进行处理。

```
    @Override
protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletR
        throws Exception {
   //1、是否是表单提交？ 该方法实现为（"POST".equals(request.getMethod())），即POST表示表单提
    if (isFormSubmission(request)) {
        try {
            Object command = getCommand(request);
            ServletRequestDataBinder binder = bindAndValidate(request, command);
            BindException errors = new BindException(binder.getBindingResult());
            //表单提交应该放到该方法实现
            return processFormSubmission(request, response, command, errors);
        }
        catch (HttpSessionRequiredException ex) {
            //省略部分代码
            return handleInvalidSubmit(request, response);
        }
    }
    else {
        //2、表示是表单展示，该方法又转调showForm方法，因此我们需要覆盖showForm来完成表单展示
        return showNewForm(request, response);
    }
}
```

bindOnNewForm：是否在进行表单展示时绑定请求参数到表单对象，默认false，不绑定；

sessionForm：session表单模式，如果开启（true）则会将表单对象放置到session中，从而可以跨越多次请求保证数据不丢失（多步骤表单常使用该方式，详解AbstractWizardFormController），默认false；

Object formBackingObject(HttpServletRequest request) ：提供给表单展示时使用的表单对象（form object表单要展示的默认数据），默认通过commandName暴露到请求给展示表单；

Map referenceData(HttpServletRequest request, Object command, Errors errors)：展示表单时需要的一些引用数据（比如用户注册，可能需要选择工作地点，这些数据可以通过该方法提供），如：

```
protected Map referenceData(HttpServletRequest request) throws Exception {
             Map model = new HashMap();
             model.put("cityList", cityList);
             return model;
}
```

这样就可以在表单展示页面获取cityList数据。

SimpleFormController继承该类，而且提供了更简单的表单流程控制。

4.9、SimpleFormController

提供了更好的两步表单支持：

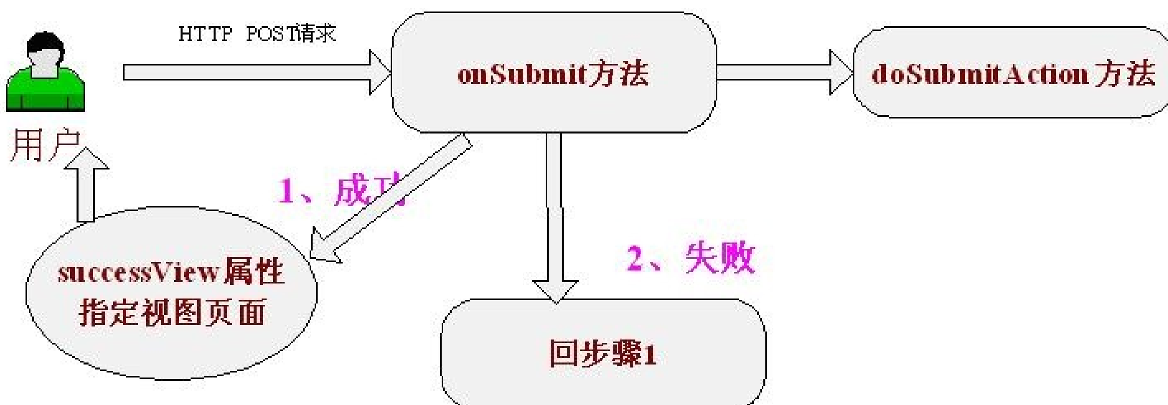
1、准备要展示的数据，并到表单展示页面；

2、提交数据数据进行处理。

第一步，展示：



第二步，提交表单：



接下来咱们写一个用户注册的例子学习一下：

（**1**、控制器

```
package cn.javass.chapter4.web.controller;
//省略import
public class RegisterSimpleFormController extends SimpleFormController {
    public RegisterSimpleFormController() {
        setCommandClass(UserModel.class); //设置命令对象实现类
        setCommandName("user");//设置命令对象的名字
    }
    //form object 表单对象，提供展示表单时的表单数据（使用commandName放入请求）
    protected Object formBackingObject(HttpServletRequest request) throws Exception {
        UserModel user = new UserModel();
        user.setUsername("请输入用户名");
        return user;
    }
    //提供展示表单时需要的一些其他数据
    protected Map referenceData(HttpServletRequest request) throws Exception {
        Map map = new HashMap();
        map.put("cityList", Arrays.asList("山东", "北京", "上海"));
        return map;
    }
    protected void doSubmitAction(Object command) throws Exception {
        UserModel user = (UserModel) command;
        //TODO 调用业务对象处理
        System.out.println(user);
    }
}
```

setCommandClass和**setCommandName**：分别设置了命令对象的实现类和名字；

formBackingObject和**referenceData**：提供了表单展示需要的视图；

doSubmitAction：用于执行表单提交动作，由onSubmit方法调用，如果不需要请求/响应对象或进行数据验证，可以直接使用doSubmitAction方法进行功能处理。

（**2**、**spring**配置（**chapter4-servlet.xml**）

```
<bean name="/simpleForm"
class="cn.javass.chapter4.web.controller.RegisterSimpleFormController">
        <property name="formView" value="register"/>
        <property name="successView" value="redirect:/success"/>
</bean>
<bean name="/success" class="cn.javass.chapter4.web.controller.SuccessController"/>
```

formView：表示展示表单时显示的页面；

successView：表示处理成功时显示的页面；“*redirect:/success*”表示成功处理后重定向到/success控制器；防止表单重复提交；

“*/success*” bean的作用是显示成功页面，此处就不列举了。

（**3**、视图页面

```
<!-- register.jsp 注册展示页面-->
<form method="post">
username:<input type="text" name="username" value="${user.username}"><br/>
password:<input type="password" name="username"><br/>
city:<select>
  <c:forEach items="${cityList }" var="city">
   <option>${city}</option>
  </c:forEach>
</select><br/>
<input type="submit" value="注册"/>
</form>
```

此处可以使用${user.username}获取到formBackingObject设置的表单对象、使用${cityList}获取referenceData设置的表单支持数据；

到此一个简单的两步表单到此结束，但这个表单有重复提交表单的问题，而且表单对象到页面的绑定是通过手工绑定的，后边我们会学习spring标签库（提供自动绑定表单对象到页面）。

4.10、CancellableFormController

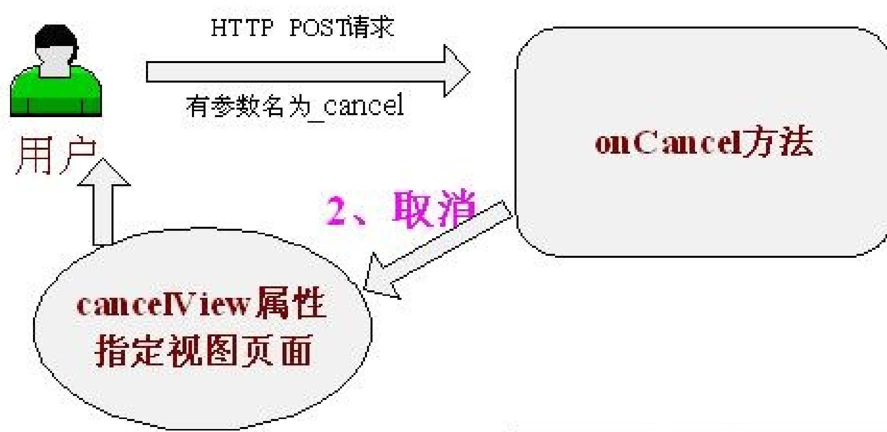
一个可取消的表单控制器，继承SimpleFormController，额外提供取消表单功能。

1、表单展示：和SimpleFormController一样；

2、表单取消：和SimpleFormController一样；

3、表单成功提交：取消功能处理方法为：onCancel(Object command)，而且默认返回cancelView属性指定的逻辑视图名。

那如何判断是取消呢？如果请求中有参数名为"_cancel"的参数，则表示表单取消。也可以通过cancelParamKey来修改参数名（如"_cancel.x"等）。



示例：

（**1**、控制器

复制RegisterSimpleFormController一份命名为CanCancelRegisterSimpleFormController，添加取消功能处理方法实现：

```
@Override
protected ModelAndView onCancel(Object command) throws Exception {
    UserModel user = (UserModel) command;
    //TODO 调用业务对象处理
    System.out.println(user);
    return super.onCancel(command);
}
```

onCancel：在该功能方法内实现取消逻辑，父类的onCancel方法默认返回cancelView属性指定的逻辑视图名。

（**2**、**spring**配置（**chapter4-servlet.xml**）

```
<bean name="/canCancelForm"
class="cn.javass.chapter4.web.controller.CanCancelRegisterSimpleFormController">
        <property name="formView" value="register"/>
        <property name="successView" value="redirect:/success"/>
        <property name="cancelView" value="redirect:/cancel"/>
</bean>
<bean name="/cancel" class="cn.javass.chapter4.web.controller.CancelController"/>
```

cancelParamKey：用于判断是否是取消的请求参数名，默认是_cancel，即如果请求参数数据中含有名字_cancel则表示是取消，将调用onCancel功能处理方法；

cancelView：表示取消时时显示的页面；"*redirect:/cancel*"表示成功处理后重定向到/cancel控制器；防止表单重复提交；

"*/cancel*" bean的作用是显示取消页面，此处就不列举了（详见代码）。

（**3**、视图页面（修改**register.jsp**）

```
<input type="submit" name="_cancel" value="取消"/>
```

该提交按钮的作用是取消，因为name="*cancel*"_，即请求后会有一个名字为_cancel的参数，因此会执行onCancel功能处理方法。

（**4**、测试：

在浏览器输入"http://localhost:9080/springmvc-chapter4/canCancelForm"，则首先到展示视图页面，点击"取消按钮"将重定向到"http://localhost:9080/springmvc-chapter4/cancel"，说明取消成功了。

实际项目可能会出现比如一些网站的完善个人资料都是多个页面（即多步），那应该怎么实现呢？接下来让我们看一下spring Web MVC提供的对多步表单的支持类AbstractWizardFormController。

[

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

4.11、AbstractWizardFormController

向导控制器类提供了多步骤（向导）表单的支持（如完善个人资料时分步骤填写基本信息、工作信息、学校信息等）

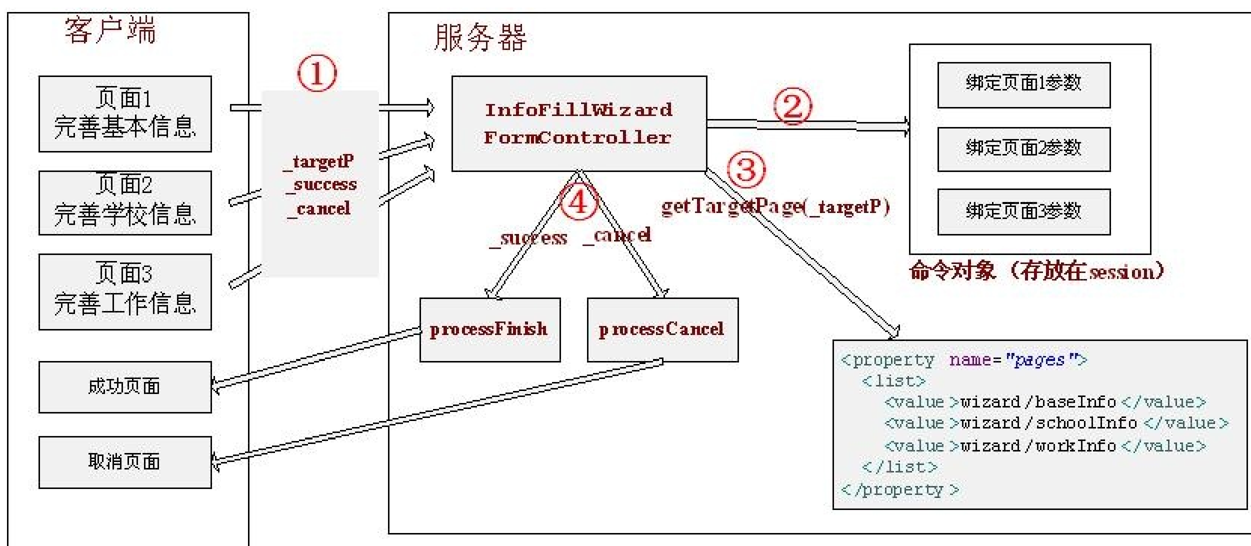
假设现在做一个完善个人信息的功能，分三个页面展示：

1、页面1完善基本信息；

2、页面2完善学校信息；

3、页面3完善工作信息。

这里我们要注意的是当用户跳转到页面2时页面1的信息是需要保存起来的，还记得AbstractFormController中的sessionForm吗？如果为true则表单数据存放到session中，哈哈，AbstractWizardFormController就是使用了这个特性。



向导中的页码从**0**开始；

PARAM_TARGET = "_target"：

用于选择向导中的要使用的页面参数名前缀，如“_target0”则选择第0个页面显示，即图中的“wizard/baseInfo”，以此类推，如“_target1”将选择第1页面，要得到的页码为去除前缀“_target”后的数字即是；

PARAM_FINISH = "_finish"：

如果请求参数中有名为“_finish”的参数，表示向导成功结束，将会调用processFinish方法进行完成时的功能处理；

PARAM_CANCEL = "_cancel"：

如果请求参数中有名为“_cancel”的参数，表示向导被取消，将会调用processCancel方法进行取消时的功能处理；

向导中的命令对象：

向导中的每一个步骤都会把相关的参数绑定到命令对象，该表单对象默认放置在session中，从而可以跨越多次请求得到该命令对象。

接下来具体看一下如何使用吧。

（**1**、修改我们的模型数据以支持多步骤提交：

```
1. public class UserModel {
2. private String username;
3. private String password;
4. private String realname; //真实姓名
5. private WorkInfoModel workInfo;
6. private SchoolInfoModel schoolInfo;
7. //省略getter/setter
8. }
```

```
1. public class SchoolInfoModel {
2. private String schoolType; //学校类型：高中、中专、大学
3. private String schoolName; //学校名称
4. private String specialty; //专业
5. //省略getter/setter
6. }
```

```
1. public class WorkInfoModel {
2. private String city; //所在城市
3. private String job; //职位
4. private String year; //工作年限
5. //省略getter/setter
6. }
```

（**2**、控制器

```
1. package cn.javass.chapter4.web.controller;
2. //省略import
3. public class InfoFillWizardFormController extends AbstractWizardFormController {
```

```
4. public InfoFillWizardFormController() {
5. setCommandClass(UserModel.class);
6. setCommandName("user");
7. }
8. protected Map referenceData(HttpServletRequest request, int page) throws Exception {
9. Map map = new HashMap();
10. if(page==1) { //如果是填写学校信息页 需要学校类型信息
11. map.put("schoolTypeList", Arrays.asList("高中", "中专", "大学"));
12. }
13. if(page==2) {//如果是填写工作信息页 需要工作城市信息
14. map.put("cityList", Arrays.asList("济南", "北京", "上海"));
15. }
16. return map;
17. }
18. protected void validatePage(Object command, Errors errors, int page) {
19. //提供每一页数据的验证处理方法
20. }
21. protected void postProcessPage(HttpServletRequest request, Object command, Errors
    errors, int page) throws Exception {
22. //提供给每一页完成时的后处理方法
23. }
24. protected ModelAndView processFinish(HttpServletRequest req, HttpServletResponse
    resp, Object command, BindException errors) throws Exception {
25. //成功后的处理方法
26. System.out.println(command);
27. return new ModelAndView("redirect:/success");
28. }
29. protected ModelAndView processCancel(HttpServletRequest request,
    HttpServletResponse response, Object command, BindException errors) throws
    Exception {
30. //取消后的处理方法
31. System.out.println(command);
32. return new ModelAndView("redirect:/cancel");
33. }
34. }
```

page页码：是根据请求中以“_target”开头的参数名来确定的，如“_target0”，则页码为0；

referenceData：提供每一页需要的表单支持对象，如完善学校信息需要学校类型，page页码从0开始（而且根据请求参数中以“_target”开头的参数来确定当前页码，如_target1，则page=1）；

validatePage：验证当前页的命令对象数据，验证应根据page页码来分步骤验证；

postProcessPage：验证成功后的后处理；

processFinish：成功时执行的方法，此处直接重定向到/success控制器（详见CancelController）；

processCancel：取消时执行的方法，此处直接重定向到/cancel控制器（详见SuccessController）；

其他需要了解：

allowDirtyBack和**allowDirtyForward**：决定在当前页面验证失败时，是否允许向导前移和后退，默认false不允许；

onBindAndValidate(HttpServletRequest request, Object command, BindException errors, int page)：允许覆盖默认的绑定参数到命令对象和验证流程。

（**3**、**spring**配置文件（**chapter4-servlet.xml**）

```
1. <bean name="/infoFillWizard"
2. class="cn.javass.chapter4.web.controller.InfoFillWizardFormController">
3. <property name="pages">
4. <list>
5. <value>wizard/baseInfo</value>
6. <value>wizard/schoolInfo</value>
7. <value>wizard/workInfo</value>
8. </list>
9. </property>
10. </bean>
```

pages：表示向导中每一个步骤的逻辑视图名，当InfoFillWizardFormController的page=0，则将会选择"wizard/baseInfo"，以此类推，从而可以按步骤选择要展示的视图。

（**4**、向导中的每一步视图

（**4.1**、基本信息页面**(**第一步**) baseInfo.jsp**：

```
1. <form method="post">
2. 真实姓名:<input type="text" name="realname" value="${user.realname}"><br/>
3. <input type="submit" name="_target1" value="下一步"/>
4. </form>
```

当前页码为**0**；

name="_target1"：表示向导下一步要显示的页面的页码为1；

（**4.2**、学校信息页面**(**第二步**) schoolInfo.jsp**：

```
1.  <form method="post">
2.  学校类型：<select name="schoolInfo.schoolType">
3.  <c:forEach items="${schoolTypeList }" var="schoolType">
4.  <option value="${schoolType }"
5.  <c:if test="${user.schoolInfo.schoolType eq schoolType}">
6.  selected="selected"
7.  </c:if>
8.  >
9.  ${schoolType}
10. </option>
11. </c:forEach>
12. </select><br/>
13. 学校名称：<input type="text" name="schoolInfo.schoolName"
    value="${user.schoolInfo.schoolName}"/><br/>
14. 专业：<input type="text" name="schoolInfo.specialty"
    value="${user.schoolInfo.specialty}"/><br/>
15. <input type="submit" name="_target0" value="上一步"/>
16. <input type="submit" name="_target2" value="下一步"/>
17. </form>
```

（**4.3**、工作信息页面**(**第三步**) workInfo.jsp**：

```
1.  <form method="post">
2.  所在城市：<select name="workInfo.city">
3.  <c:forEach items="${cityList }" var="city">
4.  <option value="${city }"
5.  <c:if test="${user.workInfo.city eq city}">selected="selected"</c:if>
6.  >
7.  ${city}
8.  </option>
9.  </c:forEach>
10. </select><br/>
11. 职位：<input type="text" name="workInfo.job" value="${user.workInfo.job}"/><br/>
12. 工作年限：<input type="text" name="workInfo.year" value="${user.workInfo.year}"/>
    <br/>
13. <input type="submit" name="_target1" value="上一步"/>
14. <input type="submit" name="_finish" value="完成"/>
15. <input type="submit" name="_cancel" value="取消"/>
16. </form>
```

当前页码为**2**；

name="_target1"：上一步，表示向导上一步要显示的页面的页码为1；

name="_finish"：向导完成，表示向导成功，将会调用向导控制器的**processFinish**方法；

name="_cancel"：向导取消，表示向导被取消，将会调用向导控制器的**processCancel**方法；

到此向导控制器完成，此处的向导流程比较简单，如果需要更复杂的页面流程控制，可以选择使用Spring Web Flow框架。

4.12、ParameterizableViewController

参数化视图控制器，不进行功能处理（即静态视图），根据参数的逻辑视图名直接选择需要展示的视图。

```
1. <bean name="/parameterizableView"
2. class="org.springframework.web.servlet.mvc.ParameterizableViewController">
3. <property name="viewName" value="success"/>
4. </bean>
```

该控制器接收到请求后直接选择参数化的视图，这样的好处是在配置文件中配置，从而避免程序的硬编码，比如像帮助页面等不需要进行功能处理，因此直接使用该控制器映射到视图。

4.13、AbstractUrlViewController

提供根据请求URL路径直接转化为逻辑视图名的支持基类，即不需要功能处理，直接根据URL计算出逻辑视图名，并选择具体视图进行展示：

urlDecode：是否进行url解码，不指定则默认使用服务器编码进行解码（如Tomcat默认ISO-8859-1）；

urlPathHelper：用于解析请求路径的工具类，默认为org.springframework.web.util.UrlPathHelper。

UrlFilenameViewController是它的一个实现者，因此我们应该使用UrlFilenameViewController。

4.14、UrlFilenameViewController

将请求的URL路径转换为逻辑视图名并返回的转换控制器，即不需要功能处理，直接根据URL计算出逻辑视图名，并选择具体视图进行展示：

根据请求URL路径计算逻辑视图名；

```
1. <bean name="/index1/*"
2. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
3. <bean name="/index2/**"
4. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
5. <bean name="/*.html"
6. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
7. <bean name="/index3/*.html"
8. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
```

/index1/*：可以匹配/index1/demo，但不匹配/index1/demo/demo，如/index1/demo逻辑视图名为demo；

/index2/：**可以匹配/index2路径下的所有子路径，如匹配/index2/demo，或/index2/demo/demo，“/index2/demo”的逻辑视图名为demo，而“/index2/demo/demo”逻辑视图名为demo/demo；

/*.html：可以匹配如/abc.html，逻辑视图名为abc，后缀会被删除（不仅仅可以是html）；

/index3/*.html：可以匹配/index3/abc.html，逻辑视图名也是abc;

上述模式为Spring Web MVC使用的Ant-style 模式进行匹配的：

1. ？匹配一个字符，如/index? 可以匹配 /index1 ， 但不能匹配 /index 或 /index12
2.
   - 匹配零个或多个字符，如/index1/*，可以匹配/index1/demo，但不匹配/index1/demo/demo
3. 匹配零个或多个路径，如**/index2/**：可以匹配/index2路径下的所有子路径，如匹配/index2/demo，或/index2/demo/demo
4. 如果我有如下模式，那Spring该选择哪一个执行呢？当我的请求为“/long/long”时如下所示：
5. /long/long
6. /long/**/abc
7. /long/**
8. /**
9. Spring的AbstractUrlHandlerMapping使用：最长匹配优先；
10. 如请求为“/long/long” 将匹配第一个“/long/long”，但请求“/long/acd” 则将匹配 “/long/**”**，如请求**“/long/aa/abc”**则匹配**“/long/**/abc”，如请求“/abc”则将匹配“/**”

UrlFilenameViewController还提供了如下属性：

prefix：生成逻辑视图名的前缀；

suffix：生成逻辑视图名的后缀；

```
1. protected String postProcessViewName(String viewName) {
2. return getPrefix() + viewName + getSuffix();
3. }
```

```
1. <bean name="/*.htm"
   class="org.springframework.web.servlet.mvc.UrlFilenameViewController">
2. <property name="prefix" value="test"/>
3. <property name="suffix" value="test"/>
4. </bean>
```

当**prefix="test"**，**suffix="test"**，如上所示的**/*.htm**：可以匹配如/abc.htm，但逻辑视图名将变为testabctest。

私塾在线学习网原创内容（http://sishuok.com）

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/5254.html】

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

4.12、ParameterizableViewController

参数化视图控制器，不进行功能处理（即静态视图），根据参数的逻辑视图名直接选择需要展示的视图。

```
1. <bean name="/parameterizableView"
2. class="org.springframework.web.servlet.mvc.ParameterizableViewController">
3. <property name="viewName" value="success"/>
4. </bean>
```

该控制器接收到请求后直接选择参数化的视图，这样的好处是在配置文件中配置，从而避免程序的硬编码，比如像帮助页面等不需要进行功能处理，因此直接使用该控制器映射到视图。

4.13、AbstractUrlViewController

提供根据请求URL路径直接转化为逻辑视图名的支持基类，即不需要功能处理，直接根据URL计算出逻辑视图名，并选择具体视图进行展示：

urlDecode：是否进行url解码，不指定则默认使用服务器编码进行解码（如Tomcat默认ISO-8859-1）；

urlPathHelper：用于解析请求路径的工具类，默认为org.springframework.web.util.UrlPathHelper。

UrlFilenameViewController是它的一个实现者，因此我们应该使用UrlFilenameViewController。

4.14、UrlFilenameViewController

将请求的URL路径转换为逻辑视图名并返回的转换控制器，即不需要功能处理，直接根据URL计算出逻辑视图名，并选择具体视图进行展示：

根据请求URL路径计算逻辑视图名；

```
1. <bean name="/index1/*"
2. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
3. <bean name="/index2/**"
```

4. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
5. <bean name="/*.html"
6. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
7. <bean name="/index3/*.html"
8. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

/index1/*：可以匹配/index1/demo，但不匹配/index1/demo/demo，如/index1/demo逻辑视图名为demo；

/index2/：**可以匹配/index2路径下的所有子路径，如匹配/index2/demo，或/index2/demo/demo，“/index2/demo”的逻辑视图名为demo，而“/index2/demo/demo”逻辑视图名为demo/demo；

/*.html：可以匹配如/abc.html，逻辑视图名为abc，后缀会被删除（不仅仅可以是html）；

/index3/*.html：可以匹配/index3/abc.html，逻辑视图名也是abc;

上述模式为Spring Web MVC使用的Ant-style 模式进行匹配的：

1. ? 匹配一个字符，如/index? 可以匹配 /index1 ， 但不能匹配 /index 或 /index12
2.
    - 匹配零个或多个字符，如/index1/*，可以匹配/index1/demo，但不匹配/index1/demo/demo
3. 匹配零个或多个路径，如**/index2/**：可以匹配/index2路径下的所有子路径，如匹配/index2/demo，或/index2/demo/demo

4. 如果我有如下模式，那Spring该选择哪一个执行呢？当我的请求为“/long/long”时如下所示：

5. /long/long
6. /long/**/abc
7. /long/**
8. /**
9. Spring的AbstractUrlHandlerMapping使用：最长匹配优先；
10. 如请求为“/long/long” 将匹配第一个“/long/long”，但请求“/long/acd” 则将匹配 “/long/**”，如请求**“/long/aa/abc”**则匹配**“/long/**/abc”，如请求“/abc”则将匹配“/**”

UrlFilenameViewController还提供了如下属性：

prefix：生成逻辑视图名的前缀；

suffix：生成逻辑视图名的后缀；

1. protected String postProcessViewName(String viewName) {
2. return getPrefix() + viewName + getSuffix();
3. }

```
1. <bean name="/*.htm"
   class="org.springframework.web.servlet.mvc.UrlFilenameViewController">
2. <property name="prefix" value="test"/>
3. <property name="suffix" value="test"/>
4. </bean>
```

当**prefix="test"**，**suffix="test"**，如上所示的**/*.htm**：可以匹配如/abc.htm，但逻辑视图名将变为testabctest。

私塾在线学习网原创内容（http://sishuok.com）

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

原创内容，转载请注明iteye http://jinnianshilongnian.iteye.com/

4.15、MultiActionController

之前学过的控制器如AbstractCommandController、SimpleFormController等一般对应一个功能处理方法（如新增），如果我要实现比如最简单的用户增删改查（CRUD Create-Read-Update-Delete），那该怎么办呢？

4.15.1 解决方案

1、每一个功能对应一个控制器，如果是CRUD则需要四个控制器，但这样我们的控制器会暴增，肯定不可取；

2、使用Spring Web MVC提供的MultiActionController，用于支持在一个控制器里添加多个功能处理方法，即将多个请求的处理方法放置到一个控制器里，这种方式不错。

4.15.2 问题

1、 MultiActionController如何将不同的请求映射不同的请求的功能处理方法呢？

Spring Web MVC提供了MethodNameResolver（方法名解析器）用于解析当前请求到需要执行的功能处理方法的方法名。默认使用InternalPathMethodNameResolver实现类，另外还提供了ParameterMethodNameResolver和PropertiesMethodNameResolver，当然我们也可以自己来实现，稍候我们仔细研究下它们是如何工作的。

2、那我们的功能处理方法应该怎么写呢？

public (ModelAndView | Map | String | void) actionName(HttpServletRequest request, HttpServletResponse response, [,HttpSession session] [,AnyObject]);

哦，原来如此，我们只需要按照如上格式写我们的功能处理方法即可；此处需要注意一下几点：

1、返回值：即模型和视图部分；

ModelAndView：模型和视图部分，之前已经见过了；

Map：只返回模型数据，逻辑视图名会根据RequestToViewNameTranslator实现类来计算，稍候讨论；

String：只返回逻辑视图名；

void：表示该功能方法直接写出response响应（如果其他返回值类型（如Map）返回null则和void进行相同的处理）；

2、**actionName**：功能方法名字；由methodNameResolver根据请求信息解析功能方法名，通过反射调用；

3、形参列表：顺序固定，“[]”表示可选，我们来看看几个示例吧：

//表示到新增页面

public ModelAndView toAdd(HttpServletRequest request, HttpServletResponse response);

//表示新增表单提交，在最后可以带着命令对象

public ModelAndView add(HttpServletRequest request, HttpServletResponse response, UserModel user);

//列表，但只返回模型数据，视图名会通过**RequestToViewNameTranslator**实现来计算

public Map list(HttpServletRequest request, HttpServletResponse response);

//文件下载，返回值类型为**void**，表示该功能方法直接写响应

public void fileDownload(HttpServletRequest request, HttpServletResponse response)

//第三个参数可以是**session**

public ModelAndView sessionWith(HttpServletRequest request, HttpServletResponse response, HttpSession session);

//如果第三个参数是**session**，那么第四个可以是命令对象，顺序必须是如下顺序

public void sessionAndCommandWith(HttpServletRequest request, HttpServletResponse response, HttpSession session, UserModel user)

4、异常处理方法，MultiActionController提供了简单的异常处理，即在请求的功能处理过程中遇到异常会交给异常处理方法进行处理，式如下所示：

public ModelAndView anyMeaningfulName(HttpServletRequest request, HttpServletResponse response, ExceptionClass exception)

MultiActionController会使用最接近的异常类型来匹配对应的异常处理方法，示例如下所示：

//处理**PayException**

public ModelAndView processPayException(HttpServletRequest request, HttpServletResponse response, PayException ex)

//处理**Exception**

public ModelAndView processException(HttpServletRequest request, HttpServletResponse response, Exception ex)

4.15.3 MultiActionController类实现

类定义：public class MultiActionController extends AbstractController implements LastModified ，继承了AbstractController，并实现了LastModified接口，默认返回-1；

核心属性：

delegate：功能处理的委托对象，即我们要调用请求处理方法所在的对象，默认是this；

methodNameResolver：功能处理方法名解析器，即根据请求信息来解析需要执行的delegate的功能处理方法的方法名。

核心方法：

```
//判断方法是否是功能处理方法
private boolean isHandlerMethod(Method method) {
    //得到方法返回值类型
    Class returnType = method.getReturnType();
    //返回值类型必须是ModelAndView、Map、String、void中的一种，否则不是功能处理方法
    if (ModelAndView.class.equals(returnType) || Map.class.equals(returnType) || String.c
            void.class.equals(returnType)) {
        Class[] parameterTypes = method.getParameterTypes();
        //功能处理方法参数个数必须>=2，且第一个是HttpServletRequest类型、第二个是HttpServletRespo
        //且不能Controller接口的handleRequest(HttpServletRequest request, HttpServletRespon
        return (parameterTypes.length >= 2 &&
                HttpServletRequest.class.equals(parameterTypes[0]) &&
                HttpServletResponse.class.equals(parameterTypes[1]) &&
                !("handleRequest".equals(method.getName()) && parameterTypes.length == 2)
    }
    return false;
}
```

```
//是否是异常处理方法
private boolean isExceptionHandlerMethod(Method method) {
    //异常处理方法必须是功能处理方法 且 参数长度为3、第三个参数类型是Throwable子类
    return (isHandlerMethod(method) &&
            method.getParameterTypes().length == 3 &&
            Throwable.class.isAssignableFrom(method.getParameterTypes()[2]));
}
```

```
private void registerHandlerMethods(Object delegate) {
    //缓存Map清空
    this.handlerMethodMap.clear();
    this.lastModifiedMethodMap.clear();
    this.exceptionHandlerMap.clear();

    //得到委托对象的所有public方法
    Method[] methods = delegate.getClass().getMethods();
    for (Method method : methods) {
        //验证是否是异常处理方法，如果是放入exceptionHandlerMap缓存map
        if (isExceptionHandlerMethod(method)) {
            registerExceptionHandlerMethod(method);
        }
        //验证是否是功能处理方法，如果是放入handlerMethodMap缓存map
        else if (isHandlerMethod(method)) {
            registerHandlerMethod(method);
            registerLastModifiedMethodIfExists(delegate, method);
        }
    }
}
```

```
protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletRespo
        throws Exception {
    try {
        //1、使用methodNameResolver 方法名解析器根据请求解析到要执行的功能方法的方法名
        String methodName = this.methodNameResolver.getHandlerMethodName(request);
        //2、调用功能方法（通过反射调用，此处就粘贴代码了）
        return invokeNamedMethod(methodName, request, response);
    }
    catch (NoSuchRequestHandlingMethodException ex) {
        return handleNoSuchRequestHandlingMethod(ex, request, response);
    }
}
```

接下来，我们看一下MultiActionController如何使用`MethodNameResolver`来解析请求到功能处理方法的方法名。

4.15.4 `MethodNameResolver`

`**1、InternalPathMethodNameResolver：**``MultiActionController的默认实现，提供从请求URL路径解析功`

`**2、ParameterMethodNameResolver：**``提供从请求参数解析功能处理方法的方法名，并按照如下顺序进行解析：`

`（1、 ``methodParamNames：``根据请求的参数名解析功能方法名（功能方法名和参数名同名）；`

```
<property name="methodParamNames" value="list,create,update"/>
```

`如上配置时，如果请求中含有参数名list、create、update时，则功能处理方法名为list、create、update，这种方`

`ParameterMethodNameResolver也考虑到图片提交按钮提交问题：`

`<input type="image" name="list"> 和submit类似可以提交表单，单击该图片后会发送两个参数“list.x`

```
for (String suffix : SUBMIT_IMAGE_SUFFIXES)  {//SUBMIT_IMAGE_SUFFIXES {“.x”, “.y”}
    if (request.getParameter(name + suffix) != null) {// name是我们配置的methodParamNames
        return true;
    }
}
```

（2、paramName：``根据请求参数名的值解析功能方法名，默认的参数名是action，即请求的参数中含有“action=que

（3、logicalMappings：``逻辑功能方法名到真实功能方法名映射，如下所示：

```
        <property name="logicalMappings">
            <props>
                <prop key="doList">list</prop>
            </props>
        </property>
```

即如果步骤1或2解析出逻辑功能方法名为doList（逻辑的），将会被重新映射为list功能方法名（真正执行的）。

（4、defaultMethodName：``默认的方法名，当以上策略失败时默认调用的方法名。

3、PropertiesMethodNameResolver：``提供自定义的从请求URL解析功能方法的方法名，使用一组用户自定义

Properties对象存放，具体配置示例如下：

```
<bean id="propertiesMethodNameResolver"
class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
    <property name="mappings">
        <props>
              <prop key="/create">create</prop>
              <prop key="/update">update</prop>
              <prop key="/delete">delete</prop>
              <prop key="/list">list</prop>
              <!-- 默认的行为 -->
              <prop key="/**">list</prop>
        </props>
    </property>
</bean>
```

对于/create请求将调用create方法，Spring内部使用PathMatcher进行匹配（默认实现是AntPathMatcher）。

4.15.5 RequestToViewNameTranslator

用于直接将请求转换为逻辑视图名。默认实现为 DefaultRequestToViewNameTranslator。

1、DefaultRequestToViewNameTranslator：``将请求URL转换为逻辑视图名，默认规则如下：

http://localhost:9080/web上下文/list -------> 逻辑视图名为list

http://localhost:9080/web上下文/list.html -------> 逻辑视图名为list(默认删除扩展名)

http://localhost:9080/web上下文/user/list.html -------> 逻辑视图名为user/list

4.15.6 示例

（1、控制器UserController

```
package cn.javass.chapter4.web.controller;
//省略import
public class UserController extends MultiActionController {
    //用户服务类
    private UserService userService;
    //逻辑视图名 通过依赖注入方式注入，可配置
    private String createView;
    private String updateView;
    private String deleteView;
    private String listView;
    private String redirectToListView;
    //省略setter/getter

    public String create(HttpServletRequest request, HttpServletResponse response, UserMo
        if("GET".equals(request.getMethod())) {
            //如果是get请求 我们转向 新增页面
            return getCreateView();
        }
        userService.create(user);
        //直接重定向到列表页面
        return getRedirectToListView();
    }
    public ModelAndView update(HttpServletRequest request, HttpServletResponse response,
        if("GET".equals(request.getMethod())) {
            //如果是get请求 我们转向更新页面
            ModelAndView mv = new ModelAndView();
            //查询要更新的数据
            mv.addObject("command", userService.get(user.getUsername()));
            mv.setViewName(getUpdateView());
            return mv;
        }
        userService.update(user);
        //直接重定向到列表页面
        return new ModelAndView(getRedirectToListView());
    }

    public ModelAndView delete(HttpServletRequest request, HttpServletResponse response,
        if("GET".equals(request.getMethod())) {
            //如果是get请求 我们转向删除页面
            ModelAndView mv = new ModelAndView();
            //查询要删除的数据
            mv.addObject("command", userService.get(user.getUsername()));
            mv.setViewName(getDeleteView());
            return mv;
        }
        userService.delete(user);
        //直接重定向到列表页面
        return new ModelAndView(getRedirectToListView());
    }

    public ModelAndView list(HttpServletRequest request, HttpServletResponse response) {
        ModelAndView mv = new ModelAndView();
        mv.addObject("userList", userService.list());
        mv.setViewName(getListView());
        return mv;
    }

    //如果使用委托方式，命令对象名称只能是command
    protected String getCommandName(Object command) {
        //命令对象的名字 默认command
        return "command";
    }
}
```

增删改：如果是GET请求方法，则表示到展示页面，POST请求方法表示真正的功能操作；

`** getCommandName：**`表示是命令对象名字，默认command，对于委托对象实现方式无法改变，因此我们就使用默认的吧。

`**（2、spring配置文件chapter4-servlet.xml**`

```
<bean id="userService" class="cn.javass.chapter4.service.UserService"/>
<bean name="/user/**" class="cn.javass.chapter4.web.controller.UserController">
    <property name="userService" ref="userService"/>
    <property name="createView" value="user/create"/>
    <property name="updateView" value="user/update"/>
    <property name="deleteView" value="user/delete"/>
    <property name="listView" value="user/list"/>
    <property name="redirectToListView" value="redirect:/user/list"/>
    <!-- 使用PropertiesMethodNameResolver来解析功能处理方法名 -->
    <!--property name="methodNameResolver" ref="propertiesMethodNameResolver"/-->
</bean>
```

`**userService：**用户服务类，实现业务逻辑；`

`** 依赖注入：**对于逻辑视图页面通过依赖注入方式注入，redirectToListView表示增删改成功后重定向的页面，防`

`** 默认使用InternalPathMethodNameResolver解析请求URL到功能方法名。**`

`**（3、视图页面**`

`**（3.1、list页面（WEB-INF/jsp/user/list.jsp）**`

```
<a href="${pageContext.request.contextPath}/user/create">用户新增</a><br/>
<table border="1" width="50%">
   <tr>
      <th>用户名</th>
      <th>真实姓名</th>
      <th>操作</th>
   </tr>
   <c:forEach items="${userList}" var="user">
   <tr>
      <td>${user.username }</td>
      <td>${user.realname }</td>
      <td>
          <a href="${pageContext.request.contextPath}/user/update?username=${user.usernam
          |
          <a href="${pageContext.request.contextPath}/user/delete?username=${user.usernam
      </td>
   </tr>
   </c:forEach>
</table>
```

`**（3.2、update页面（WEB-INF/jsp/user/update.jsp）**`

```
<form action="${pageContext.request.contextPath}/user/update" method="post">
用户名： <input type="text" name="username" value="${command.username}"/><br/>
真实姓名：<input type="text" name="realname" value="${command.realname}"/><br/>
<input type="submit" value="更新"/>
</form>
```

`**（4、测试：**`

默认的**InternalPathMethodNameResolver**将进行如下解析：

http://localhost:9080/springmvc-chapter4/user/list————>list方法名；

http://localhost:9080/springmvc-chapter4/user/create————>create方法名；

http://localhost:9080/springmvc-chapter4/user/update————>update功能处理方法名；

http://localhost:9080/springmvc-chapter4/user/delete————>delete功能处理方法名。

我们可以将默认的**InternalPathMethodNameResolver**改为**PropertiesMethodNameResolver**：

```
<bean id="propertiesMethodNameResolver"
class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
    <property name="mappings">
        <props>
              <prop key="/user/create">create</prop>
              <prop key="/user/update">update</prop>
              <prop key="/user/delete">delete</prop>
              <prop key="/user/list">list</prop>
              <prop key="/**">list</prop><!-- 默认的行为 -->
          </props>
    </property>
    <property name="alwaysUseFullPath" value="false"/><!-- 不使用全路径 -->
</bean>
<bean name="/user/**" class="cn.javass.chapter4.web.controller.UserController">
       <!—省略其他配置，详见配置文件-->
       <!-- 使用PropertiesMethodNameResolver来解析功能处理方法名 -->
       <property name="methodNameResolver" ref="propertiesMethodNameResolver"/>
</bean>
```

/表示默认解析到list功能处理方法。**

如上配置方式可以很好的工作，但必须继承MultiActionController，Spring Web MVC提供给我们无需继承MultiActionController实现方式，即使有委托对象方式，继续往下看吧。

4.15.7、委托方式实现

`**（1、控制器UserDelegate**`

`将UserController复制一份，改名为UserDelegate，并把继承MultiActionController去掉即可，其他无需改变。`

`**（2、spring配置文件chapter4-servlet.xml**`

```
<!—委托对象-->
<bean id="userDelegate" class="cn.javass.chapter4.web.controller.UserDelegate">
   <property name="userService" ref="userService"/>
   <property name="createView" value="user2/create"/>
   <property name="updateView" value="user2/update"/>
   <property name="deleteView" value="user2/delete"/>
   <property name="listView" value="user2/list"/>
   <property name="redirectToListView" value="redirect:/user2/list"/>
</bean>
<!—控制器对象-->
<bean name="/user2/**"
class="org.springframework.web.servlet.mvc.multiaction.MultiActionController">
<property name="delegate" ref="userDelegate"/>
    <property name="methodNameResolver" ref="parameterMethodNameResolver"/>
</bean>
```

delegate：`控制器对象通过`delegate属性指定委托对象，即实际调用delegate委托对象的功能方法。

methodNameResolver：此处我们使用ParameterMethodNameResolver解析器；

```
<!—ParameterMethodNameResolver -->
<bean id="parameterMethodNameResolver"
class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
<!-- 1、根据请求参数名解析功能方法名 -->
    <property name="methodParamNames" value="create,update,delete"/>
    <!-- 2、根据请求参数名的值解析功能方法名 -->
    <property name="paramName" value="action"/>
<!-- 3、逻辑方法名到真实方法名的映射 -->
    <property name="logicalMappings">
       <props>
           <prop key="doList">list</prop>
       </props>
    </property>
    <!—4、默认执行的功能处理方法 -->
    <property name="defaultMethodName" value="list"/>
</bean>
```

`**1、**` **methodParamNames**：create,update,delete，当请求中有参数名为这三个的将被映射为功能方法名，如“<input type="submit" name="create" value="新增"/>”提交后解析得到的功能方法名为create;

`**2、paramName：**` 当请求中有参数名为action，则将值映射为功能方法名，如“<input type="hidden" name="action" value="delete"/>”，提交后解析得到的功能方法名为delete；

`**3、logicalMappings：**` 逻辑功能方法名到真实功能方法名的映射，如：

http://localhost:9080/springmvc-chapter4/user2?action=doList；

`首先请求参数“action=doList”，则第二步解析得到逻辑功能方法名为doList；`

`本步骤会把doList再转换为真实的功能方法名list。`

`**4、defaultMethodName：**` 以上步骤如果没有解析到功能处理方法名，默认执行的方法名。

`**（3、视图页面**`

`**（3.1、list页面（WEB-INF/jsp/user2/list.jsp）**`

```
<a href="${pageContext.request.contextPath}/user2?action=create">用户新增</a><br/>
<table border="1" width="50%">
   <tr>
      <th>用户名</th>
      <th>真实姓名</th>
      <th>操作</th>
   </tr>
   <c:forEach items="${userList}" var="user">
   <tr>
      <td>${user.username }</td>
      <td>${user.realname }</td>
      <td>
          <a href="${pageContext.request.contextPath}/user2?action=update&username=${user
          |
          <a href="${pageContext.request.contextPath}/user2?action=delete&username=${user
      </td>
   </tr>
   </c:forEach>
</table>
```

`**（3.2、update页面（WEB-INF/jsp/user2/update.jsp）**`

```
<form action="${pageContext.request.contextPath}/user2" method="post">
<input type="hidden" name="action" value="update"/>
用户名： <input type="text" name="username" value="${command.username}"/><br/>
真实姓名：<input type="text" name="realname" value="${command.realname}"/><br/>
<input type="submit" value="更新"/>
</form>
```

`通过参数` name=*"action"* value=*"update"* `来指定要执行的功能方法名update。`

`**（3.3、create页面（WEB-INF/jsp/user2/create.jsp）**`

```
<form action="${pageContext.request.contextPath}/user2" method="post">
用户名： <input type="text" name="username" value="${command.username}"/><br/>
真实姓名：<input type="text" name="realname" value="${command.realname}"/><br/>
<input type="submit" name="create" value="新增"/>
</form>
```

`通过参数` name=*"create"* `来指定要执行的功能方法名create。`

`**（4、测试：**`

使用**ParameterMethodNameResolver**将进行如下解析：

http://localhost:9080/springmvc-chapter4/user2?create ————>create功能处理方法名（参数名映射）；

http://localhost:9080/springmvc-chapter4/user2?action=create————>create功能处理方法名（参数值映射）；

http://localhost:9080/springmvc-chapter4/user2?update ————>update功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?action=update————>update功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?delete ———>delete功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?action=delete———>delete功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?doList ———>通过logicalMappings解析为list功能处理方法。

http://localhost:9080/springmvc-chapter4/user2?action=doList———>通过logicalMappings解析为list功能处理方法。

http://localhost:9080/springmvc-chapter4/user2———>默认的功能处理方法名list（默认）。

原创内容，转载请注明iteye http://jinnianshilongnian.iteye.com/

跟着开涛学SpringMVC 第一章源代码下载

源代码请到附件中下载。

其他下载：

跟着开涛学SpringMVC 第一章源代码下载

第二章 Spring MVC入门 源代码下载

Controller接口控制器详解 源代码下载

源码下载——第四章 Controller接口控制器详解——跟着开涛学SpringMVC

源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC

目录：[

第一章 Web MVC简介 —— 跟开涛学SpringMVC](/blog/1593441 "第一章 Web MVC简介 —— 跟开涛学SpringMVC")

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第五章 处理器拦截器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

第二章 Spring MVC入门 源代码下载

源代码请到附件中下载。

其他下载：

跟着开涛学SpringMVC 第一章源代码下载

第二章 Spring MVC入门 源代码下载

Controller接口控制器详解 源代码下载

源码下载——第四章 Controller接口控制器详解——跟着开涛学SpringMVC

源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC

目录：[

第一章 Web MVC简介 —— 跟开涛学SpringMVC](/blog/1593441 "第一章 Web MVC简介 —— 跟开涛学SpringMVC")

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第五章 处理器拦截器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

第四章 Controller接口控制器详解 源代码下载

源代码请到附件中下载。

其他下载：

跟着开涛学SpringMVC 第一章源代码下载

第二章 Spring MVC入门 源代码下载

Controller接口控制器详解 源代码下载

源码下载——第四章 Controller接口控制器详解——跟着开涛学SpringMVC

源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC

目录：[

第一章 Web MVC简介 —— 跟开涛学SpringMVC](/blog/1593441 "第一章 Web MVC简介 —— 跟开涛学SpringMVC")

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第五章 处理器拦截器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第一章 Web MVC简介 —— 跟开涛学SpringMVC

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

4.16、数据类型转换和数据验证



流程：

1、首先创建数据绑定器，在此此会创建ServletRequestDataBinder类的对象，并设置messageCodesResolver（错误码解析器）；

2、提供第一个扩展点，初始化数据绑定器，在此处我们可以覆盖该方法注册自定义的PropertyEditor（请求参数——>命令对象属性的转换）；

3、进行数据绑定，即请求参数——>命令对象的绑定；

4、提供第二个扩展点，数据绑定完成后的扩展点，此处可以实现一些自定义的绑定动作；

5、验证器对象的验证，验证器通过validators注入，如果验证失败，需要把错误信息放入Errors（此处使用BindException实现）；

6、提供第三个扩展点，此处可以实现自定义的绑定/验证逻辑；

7、将errors传入功能处理方法进行处理，功能方法应该判断该错误对象是否有错误进行相应的处理。

4.16.1、数据类型转换

请求参数（String）——>命令对象属性（可能是任意类型）的类型转换，即数据绑定时的类型转换，使用PropertyEditor实现绑定时的类型转换。

一、**Spring**内建的**PropertyEditor**如下所示：

类名	说明	默认是否注册
ByteArrayPropertyEditor	String<——>byte[]	√
ClassEditor	String<——>Class当类没有发现抛出 `IllegalArgumentException`	√
CustomBooleanEditor	String<——>Booleantrue/yes/on/1转换为true，false/no/off/0转换为false	√
CustomCollectionEditor	数组/Collection——>Collection普通值——>Collection（只包含一个对象）如String——>Collection不允许Collection——>String（单方向转换）	√
CustomNumberEditor	String<——>Number(Integer、Long、Double)	√
FileEditor	String<——>File	√
InputStreamEditor	String——>InputStream单向的，不能InputStream——>String	√
LocaleEditor	String<——>Locale，（String的形式为[语言]/[国家]/[变量]，这与Local对象的toString()方法得到的结果相同）	√
PatternEditor	String<——>Pattern	√
PropertiesEditor	String<——>java.lang.Properties	√
URLEditor	String<——>URL	√
StringTrimmerEditor	一个用于trim 的 String类型的属性编辑器如默认删除两边的空格，charsToDelete属性：可以设置为其他字符emptyAsNull属性：将一个空字符串转化为null值的选项。	×
CustomDateEditor	String<——>java.util.Date	×

二、**Spring**内建的**PropertyEditor**支持的属性（符合**JavaBean**规范）操作：

表达式	设值/取值说明
username	属性username设值方法setUsername()/取值方法getUsername() 或 isUsername()
schoolInfo.schoolType	属性schoolInfo的嵌套属性schoolType设值方法getSchoolInfo().setSchoolType()/取值方法getSchoolInfo().getSchoolType()
hobbyList[0]	属性hobbyList的第一个元素索引属性可能是一个数组、列表、其它天然有序的容器。
map[key]	属性map（java.util.Map类型）map中key对应的值

三、示例：

接下来我们写自定义的属性编辑器进行数据绑定：

（**1**、模型对象：

```
package cn.javass.chapter4.model;
//省略import
public class DataBinderTestModel {
    private String username;
    private boolean bool;//Boolean值测试
    private SchoolInfoModel schooInfo;
    private List hobbyList;//集合测试，此处可以改为数组/Set进行测试
    private Map map;//Map测试
    private PhoneNumberModel phoneNumber;//String->自定义对象的转换测试
    private Date date;//日期类型测试
    private UserState state;//String——>Enum类型转换测试
    //省略getter/setter
}

package cn.javass.chapter4.model;
//如格式010-12345678
public class PhoneNumberModel {
    private String areaCode;//区号
    private String phoneNumber;//电话号码
    //省略getter/setter
}
```

(2、**PhoneNumber**属性编辑器

前台输入如010-12345678自动转换为PhoneNumberModel。

```
package cn.javass.chapter4.web.controller.support.editor;
//省略import
public class PhoneNumberEditor extends PropertyEditorSupport {
    Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        if(text == null || !StringUtils.hasLength(text)) {
            setValue(null); //如果没值，设值为null
        }
        Matcher matcher = pattern.matcher(text);
        if(matcher.matches()) {
            PhoneNumberModel phoneNumber = new PhoneNumberModel();
            phoneNumber.setAreaCode(matcher.group(1));
            phoneNumber.setPhoneNumber(matcher.group(2));
            setValue(phoneNumber);
        } else {
            throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-1234
        }
    }
    @Override
    public String getAsText() {
        PhoneNumberModel phoneNumber = ((PhoneNumberModel)getValue());
        return phoneNumber == null ? "" : phoneNumber.getAreaCode() + "-" + phoneNumber.g
    }
}
```

PropertyEditorSupport：一个PropertyEditor的支持类；

setAsText：表示将String——>PhoneNumberModel，根据正则表达式进行转换，如果转换失败抛出异常，则接下来的验证器会进行验证处理；

getAsText：表示将PhoneNumberModel——>String。

（**3**、控制器

需要在控制器注册我们自定义的属性编辑器。

此处我们使用AbstractCommandController，因为它继承了BaseCommandController，拥有绑定流程。

```
package cn.javass.chapter4.web.controller;
//省略import
public class DataBinderTestController extends AbstractCommandController {
    public DataBinderTestController() {
        setCommandClass(DataBinderTestModel.class); //设置命令对象
        setCommandName("dataBinderTest");//设置命令对象的名字
    }
    @Override
    protected ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Objec
        //输出command对象看看是否绑定正确
        System.out.println(command);
        return new ModelAndView("bindAndValidate/success").addObject("dataBinderTest", co
    }
    @Override
    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder
        super.initBinder(request, binder);
        //注册自定义的属性编辑器
        //1、日期
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        CustomDateEditor dateEditor = new CustomDateEditor(df, true);
        //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
        binder.registerCustomEditor(Date.class, dateEditor);
        //自定义的电话号码编辑器
        binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());
    }
}
```

initBinder:第一个扩展点，初始化数据绑定器，在此处我们注册了两个属性编辑器；

CustomDateEditor：自定义的日期编辑器，用于在String<——>日期之间转换；

binder.registerCustomEditor(Date.**class**, dateEditor)：表示如果命令对象是Date类型，则使用dateEditor进行类型转换；

PhoneNumberEditor：自定义的电话号码属性编辑器用于在String<——>PhoneNumberModel之间转换；

binder.registerCustomEditor(PhoneNumberModel.**class**, **new**PhoneNumberEditor())：表示如果命令对象是PhoneNumberModel类型，则使用PhoneNumberEditor进行类型转换；

`**（4、spring配置文件chapter4-servlet.xml**`

```
<bean name="/dataBind"
class="cn.javass.chapter4.web.controller.DataBinderTestController"/>
```

（**5**、视图页面（**WEB-INF/jsp/bindAndValidate/success.jsp**）

```
EL phoneNumber:${dataBinderTest.phoneNumber}<br/>
EL state:${dataBinderTest.state}<br/>
EL date:${dataBinderTest.date}<br/>
```

视图页面的数据没有预期被格式化，如何进行格式化显示呢？请参考【第七章 注解式控制器的数据验证、类型转换及格式化】。

（**6**、测试：

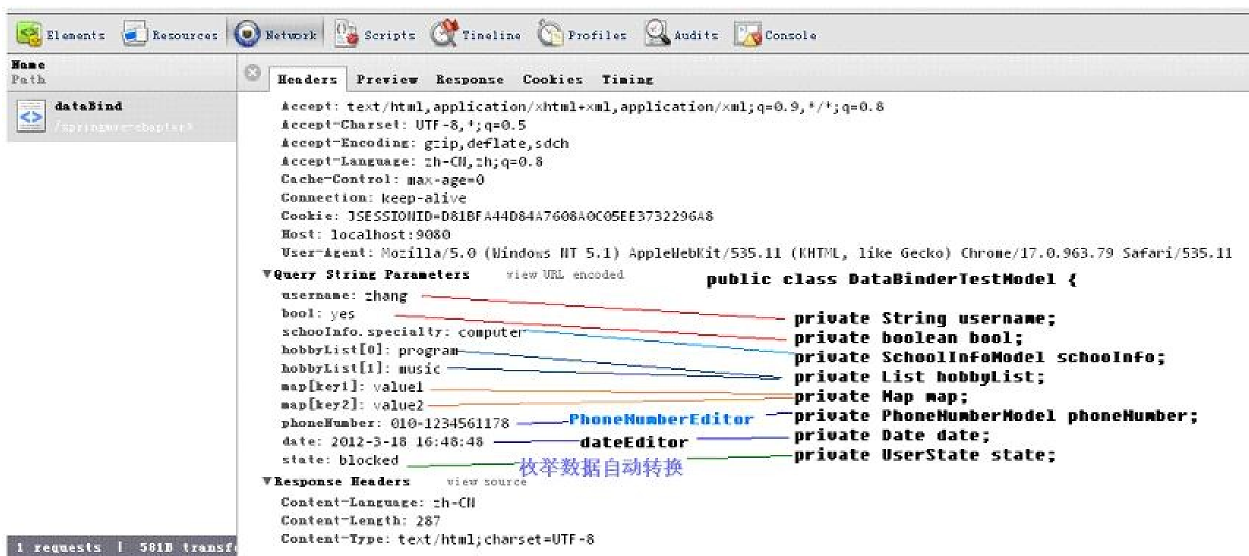
1、在浏览器地址栏输入请求的**URL**，如

http://localhost:9080/springmvc-chapter4/dataBind?username=zhang&bool=yes&schoolInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&phoneNumber=010-12345678&date=2012-3-18 16:48:48&state=blocked

2、控制器输出的内容：

DataBinderTestModel [username=zhang, bool=true, schoolInfo=SchoolInfoModel [schoolType=null, schoolName=null, specialty=computer], hobbyList=[program, music], map={key1=value1, key2=value2}, phoneNumber=PhoneNumberModel [areaCode=010, phoneNumber=12345678], date=Sun Mar 18 16:48:48 CST 2012, state=锁定]

类型转换如图所示：



四、注册**PropertyEditor**

1、使用**WebDataBinder**进行控制器级别注册**PropertyEditor**（控制器独享）

如"【三、示例】"中所使用的方式，使用WebDataBinder注册控制器级别的PropertyEditor，这种方式注册的PropertyEditor只对当前控制器独享，即其他的控制器不会自动注册这个PropertyEditor，如果需要还需要再注册一下。

2、使用 `**WebBindingInitializer批量注册**` **PropertyEditor**

如果想在多个控制器同时注册多个相同的PropertyEditor时，可以考虑使用WebBindingInitializer。

示例：

（1、实现WebBindingInitializer

```
package cn.javass.chapter4.web.controller.support.initializer;
//省略import
public class MyWebBindingInitializer implements WebBindingInitializer {
    @Override
    public void initBinder(WebDataBinder binder, WebRequest request) {
        //注册自定义的属性编辑器
        //1、日期
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        CustomDateEditor dateEditor = new CustomDateEditor(df, true);
        //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
        binder.registerCustomEditor(Date.class, dateEditor);
        //自定义的电话号码编辑器
        binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());
    }
}
```

通过实现WebBindingInitializer并通过binder注册多个PropertyEditor。

（2、修改【三、示例】中的DataBinderTestController，注释掉initBinder方法；

（3、修改chapter4-servlet.xml配置文件：

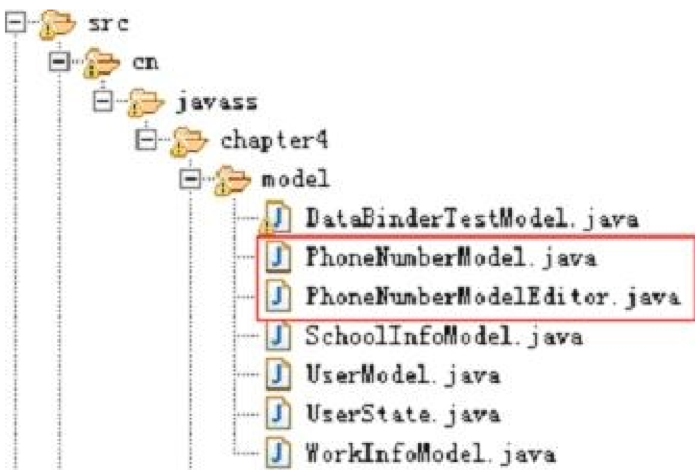
```
<!-- 注册WebBindingInitializer实现 -->
<bean id="myWebBindingInitializer" class="cn.javass.chapter4.web.controller.support.initi
<bean name="/dataBind" class="cn.javass.chapter4.web.controller.DataBinderTestController"
    <!-- 注入WebBindingInitializer实现 -->
    <property name="webBindingInitializer" ref="myWebBindingInitializer"/>
</bean>
```

（4、尝试访问"【三、示例】"中的测试URL即可成功。

使用WebBindingInitializer的好处是当你需要在多个控制器中需要同时使用多个相同的PropertyEditor可以在WebBindingInitializer实现中注册，这样只需要在控制器中注入WebBindingInitializer即可注入多个PropertyEditor。

3、全局级别注册PropertyEditor（全局共享）

只需要将我们自定义的PropertyEditor放在和你的模型类同包下即可，且你的Editor命名规则必须是"模型类名Editor"，这样Spring会自动使用标准JavaBean架构进行自动识别，如图所示：



此时我们把“DataBinderTestController”的“binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());”注释掉，再尝试访问“【三、示例】”中的测试URL即可成功。

这种方式不仅仅在使用Spring时可用，在标准的JavaBean等环境都是可用的，可以认为是全局共享的（不仅仅是Spring环境）。

PropertyEditor被限制为只能**String<——>Object**之间转换，不能**Object<——>Object**，**Spring3**提供了更强大的类型转换（**Type** `**Conversion）支持，它可以在任意对象之间进行类型转换，不仅仅是String**` **<** `**——>Object。**`

如果我在地址栏输入错误的数据，即数据绑定失败，Spring Web MVC该如何处理呢？如果我输入的数据不合法呢？如如用户名输入100个字符（超长了）那又该怎么处理呢？出错了需要错误消息，那错误消息应该是硬编码？还是可配置呢？

接下来我们来学习一下数据验证器进行数据验证吧。

私塾在线学习网原创内容（http://sishuok.com）

原创内容，转载请注明私塾在线【http://sishuok.com/forum/blogPost/list/5677.html】

第四章 Controller接口控制器详解（7 完）——跟着开涛学SpringMVC

4.16.2、数据验证

1、数据绑定失败：比如需要数字却输入了字母；

2、数据不合法：可以认为是业务错误，通过自定义验证器验证，如用户名长度必须在5-20之间，我们却输入了100个字符等；

3、错误对象：当我们数据绑定失败或验证失败后，错误信息存放的对象，我们叫错误对象，在Spring Web MVC中Errors是具体的代表者；线程不安全对象；

4、错误消息：是硬编码，还是可配置？实际工作应该使用配置方式，我们只是把错误码（errorCode）放入错误对象，在展示时读取相应的错误消息配置文件来获取要显示的错误消息(errorMessage)；

4.16.2.1、验证流程

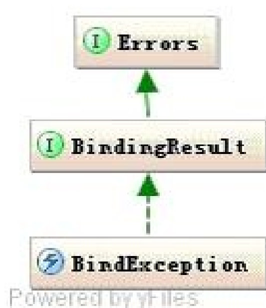


1、首先进行数据绑定验证，如果验证失败会通过MessageCodesResolver生成错误码放入Errors错误对象；

2、数据不合法验证，通过自定义的验证器验证，如果失败需要手动将错误码放入Errors错误对象；

4.16.2.2、错误对象和错误消息

错误对象的代表者是Errors接口，并且提供了几个实现者，在Spring Web MVC中我们使用的是如下实现：



相关的错误方法如下：

Errors：存储和暴露关于数据绑定错误和验证错误相关信息的接口，提供了相关存储和获取错误消息的方法：

```
package org.springframework.validation;
public interface Errors {
  //=========================全局错误消息（验证/绑定对象全局的）=============================
  //注册一个全局的错误码（）
  void reject(String errorCode);
  //注册一个全局的错误码，当根据errorCode没有找到相应错误消息时，使用defaultMessage作为错误消息
  void reject(String errorCode, String defaultMessage);
  //注册一个全局的错误码，当根据errorCode没有找到相应错误消息时（带错误参数的），使用defaultMessage作
  void reject(String errorCode, Object[] errorArgs, String defaultMessage);
  //=========================全局错误消息（验证/绑定整个对象的）=============================
  //=========================局部错误消息（验证/绑定对象字段的）=============================
  //注册一个对象字段的错误码，field指定验证失败的字段名
  void rejectValue(String field, String errorCode);
  void rejectValue(String field, String errorCode, String defaultMessage);
  void rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMess
  //=========================局部错误消息（验证/绑定对象字段的）=============================
  boolean hasErrors();      ////是否有错误
  boolean hasGlobalErrors(); //是否有全局错误
  boolean hasFieldErrors();  //是否有字段错误
  Object getFieldValue(String field); //返回当前验证通过的值，或验证失败时失败的值；
}
```

getFieldValue：可以得到验证失败的失败值，这是其他Web层框架很少支持的，这样就可以给用户展示出错时的值（而不是空或其他的默认值等）。

BindingResult：代表数据绑定的结果，继承了Errors接口。

BindException：代表数据绑定的异常，它继承Exception，并实现了BindingResult，这是内部使用的错误对象。

示例：

（**1**、控制器

```
package cn.javass.chapter4.web.controller;
//省略import
public class ErrorController extends AbstractCommandController {
        public ErrorController() {
                setCommandClass(DataBinderTestModel.class);
                setCommandName("command");
        }
        @Override
        protected ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Ob
                //表示用户名不为空
                errors.reject("username.not.empty");
                //带有默认错误消息
                errors.reject("username.not.empty1", "用户名不能为空1");
                //带有参数和默认错误消息
                errors.reject("username.length.error", new Object[]{5, 10});

                //得到错误相关的模型数据
                Map model = errors.getModel();
                return new ModelAndView("bindAndValidate/error", model);
        }
}
```

```
errors.reject("username.not.empty")：注册全局错误码“username.not.empty”，我们必须提供messageSo
```

errors.reject("username.not.empty1", "用户名不能为空**1")**：注册全局错误码“username.not.empty1”，如果从messageSource没没有找到错误码“username.not.empty1”对应的错误信息，则将显示默认消息“用户名不能为空1”；

errors.reject("username.length.error", new Object[]{5, 10})：错误码为“username.length.error”，而且错误信息需要两个参数，如我们在我们的配置文件中定义“用户名长度不合法，长度必须在{0}到{1}之间”，则实际的错误消息为“用户名长度不合法，长度必须在5到10之间”

errors.getModel()：当有错误信息时，一定将errors.getModel()放入我们要返回的ModelAndView中，以便使用里边的错误对象来显示错误信息。

`**（2、spring配置文件chapter4-servlet.xml**`

```
<bean id="messageSource"
       class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:messages"/>
    <property name="fileEncodings" value="utf-8"/>
    <property name="cacheSeconds" value="120"/>
</bean>

<bean name="/error" class="cn.javass.chapter4.web.controller.ErrorController"/>
```

messageSource：用于获取错误码对应的错误消息的，而且bean名字默认必须是messageSource。

messages.properties（需要执行NativeToAscii）

```
username.not.empty=用户名不能为空
username.length.error=用户名长度不合法，长度必须在{0}到{1}之间
```

（**3**、视图页面（**WEB-INF/jsp/bindAndValidate/error.jsp**）

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!-- 表单的默认命令对象名为command -->
<form:form commandName="command">
    <form:errors path="*"></form:errors>
</form:form>
```

form标签库：此处我们使用了spring的form标签库；

<form:form commandName="command">:表示我们的表单标签，commandName表示绑定的命令对象名字，默认为command；

<form:errors path="*"></form:errors>：表示显示错误信息的标签，如果path为"*"表示显示所有错误信息。

接下来我们来看一下 数据绑定失败和数据不合法时，如何处理。

4.16.2.3、数据绑定失败

如我们的DataBinderTestModel类：

bool：boolean类型，此时如果我们前台传入非兼容的数据，则会数据绑定失败；

date：Date类型，此时如果我们前台传入非兼容的数据，同样会数据绑定失败；

phoneNumber：自定义的PhoneNumberModel类型，如果如果我们前台传入非兼容的数据，同样会数据绑定失败。

示例：

（**1**、控制器，**DataBinderErrorTestController**。

```
package cn.javass.chapter4.web.controller;
//省略import
public class DataBinderErrorTestController extends SimpleFormController {
        public DataBinderErrorTestController() {
                setCommandClass(DataBinderTestModel.class);
                setCommandName("dataBinderTest");
        }
        @Override
        protected ModelAndView showForm(HttpServletRequest request, HttpServletResponse re
                //如果表单提交有任何错误都会再回到表单展示页面
                System.out.println(errors);
                return super.showForm(request, response, errors);
        }
        @Override
        protected void doSubmitAction(Object command) throws Exception {
                System.out.println(command); //表单提交成功（数据绑定成功）进行功能处理
    }
        @Override
        protected void initBinder(HttpServletRequest request, ServletRequestDataBinder bin
                super.initBinder(request, binder);
                //注册自定义的属性编辑器
                //1、日期
                DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
                CustomDateEditor dateEditor = new CustomDateEditor(df, true);
                //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
                binder.registerCustomEditor(Date.class, dateEditor);

                //自定义的电话号码编辑器
                binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor()
        }
}
```

此处我们使用SimpleFormController；

showForm：展示表单，当提交表单有任何数据绑定错误会再回到该方法进行表单输入（在此处我们打印错误对象）；

doSubmitAction：表单提交成功，只要当表单的数据到命令对象绑定成功时，才会执行；

`**（2、spring配置文件chapter4-servlet.xml**`

```
<bean name="/dataBindError"
class="cn.javass.chapter4.web.controller.DataBinderErrorTestController">
   <property name="formView" value="bindAndValidate/input"/>
   <property name="successView" value="bindAndValidate/success"/>
</bean>
```

（**3**、视图页面（**WEB-INF/jsp/bindAndValidate/ input.jsp**）

```
<%@ page language="java" contentType="text/html; charset=UTF-8"    pageEncoding="UTF-8"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!-- 表单的命令对象名为dataBinderTest -->
<form:form commandName="dataBinderTest">
    <form:errors path="*" cssStyle="color:red"></form:errors><br/><br/>
    bool:<form:input path="bool"/><br/>
    phoneNumber:<form:input path="phoneNumber"/><br/>
    date:<form:input path="date"/><br/>
    <input type="submit" value="提交"/>
</form:form>
```

此处一定要使用form标签库，借此我们可以看到它的强大支持（别的大部分Web框架所不具备的，展示用户验证失败的数据）。

<form:form commandName=*"dataBinderTest"*>：指定命令对象为dataBinderTest，默认command；

<form:errors path=*"*"* cssStyle=*"color:red"*></form:errors>：显示错误消息，当提交表单有错误时展示错误消息（数据绑定错误/数据不合法）；

<form:input path=*"bool"*/>：等价于（<input type='text'>），但会从命令对象中取出bool属性进行填充value属性，或如果表单提交有错误会从错误对象取出之前的错误数据（而非空或默认值）；

<input type=*"submit"* value=*"提交"*/>：spring没有提供相应的提交按钮，因此需要使用html的。

（**4**、测试

在地址栏输入如下地址：http://localhost:9080/springmvc-chapter4/dataBindError

bool: www
phoneNumber: 123
date: 123
提交

全部是错误数据，即不能绑定到我们的命令对象；

当提交表单后，我们又回到表单输入页面，而且输出了一堆错误信息

Failed to convert property value of type java.lang.String to required type boolean for property bool; nested exception is java.lang.IllegalArgumentException: Invalid boolean value [www]
Failed to convert property value of type java.lang.String to required type java.util.Date for property date; nested exception is java.lang.IllegalArgumentException: Could not parse date: Unparseable date: "123"
Failed to convert property value of type java.lang.String to required type cn.javass.chapter3.model.PhoneNumberModel for property phoneNumber; nested exception is java.lang.IllegalArgumentException: 类型转换失败，需要格式[010-12345678]，但格式是[123]

错误消息

bool: www
phoneNumber: 123
date: 123
提交

显示之前的错误的数据，而不是默认值/空，其他部分web框架所不具备的

1、错误消息不可读；

2、表单元素可以显示之前的错误的数据，而不是默认值/空；

（**5**、问题

这里最大的问题是不可读的错误消息，如何让这些错误消息可读呢？

首先我们看我们的showForm方法里输出的“errors”错误对象信息：

```
org.springframework.validation.BindException: org.springframework.validation.BeanProperty
Field error in object 'dataBinderTest' on field 'bool': rejected value [www]; codes [type
Field error in object 'dataBinderTest' on field 'date': rejected value [123]; codes [type
Field error in object 'dataBinderTest' on field 'phoneNumber': rejected value [123]; code
```

数据绑定失败（类型不匹配）会自动生成如下错误码（错误码对应的错误消息按照如下顺序依次查找）：

1、typeMismatch.命令对象名.属性名

2、typeMismatch.属性名

3、typeMismatch.属性全限定类名（包名.类名）

4、typeMismatch

⊙内部使用MessageCodesResolver解析数据绑定错误到错误码，默认DefaultMessageCodesResolver，因此想要详细了解如何解析请看其javadoc；

⊙建议使用第1个进行错误码的配置。

因此修改我们的messages.properties添加如下错误消息（需要执行NativeToAscii）：

```
typeMismatch.dataBinderTest.date=您输入的数据格式错误，请重新输入（格式：2012-03-19 22:17:17）
#typeMismatch.date=2
#typeMismatch.java.util.Date=3
#typeMismatch=4
```

再次提交表单我们会看到我们设置的错误消息：

Failed to convert property value of type java.lang.String to required type boolean for property bool; nested exception is java.lang.IllegalArgumentException: Invalid boolean value [www]
您输入的数据格式错误，请重新输入（格式：2012-03-19 22:17:17）
Failed to convert property value of type java.lang.String to required type cn.javass.chapter3.model.PhoneNumberModel for property phoneNumber; nested exception is java.lang.IllegalArgumentException: 类型转换失败，需要格式[010-12345678]，但格式是[123]

bool: www
phoneNumber: 123
date: 123
提交

到此，数据绑定错误我们介绍完了，接下来我们再看一下数据不合法错误。

4.16.2.4、数据不合法

1、比如用户名长度必须在5-20之间，而且必须以字母开头，可包含字母、数字、下划线；

2、比如注册用户时 用户名已经存在或邮箱已经存在等；

3、比如去一些论坛经常会发现，您发的帖子中包含×××屏蔽关键字等。

还有很多数据不合法的场景，在此就不罗列了，对于数据不合法，Spring Web MVC提供了两种验证方式：

◆编程式验证器验证

◆声明式验证

先从编程式验证器开始吧。

4.16.2.4.1、编程式验证器

一、验证器接口

```
package org.springframework.validation;
public interface Validator {
boolean supports(Class<?> clazz);
void validate(Object target, Errors errors);
}
```

Validator接口：验证器，编程实现数据验证的接口；

supports方法：当前验证器是否支持指定的clazz验证，如果支持返回true即可；

validate方法：验证的具体方法，target参数表示要验证的目标对象（如命令对象），errors表示验证出错后存放错误信息的错误对象。

示例：

（**1**、验证器实现

```
package cn.javass.chapter4.web.controller.support.validator;
//省略import
public class UserModelValidator implements Validator {
    private static final Pattern USERNAME_PATTERN = Pattern.compile("[a-zA-Z]\\w{4,19}");
    private static final Pattern PASSWORD_PATTERN = Pattern.compile("[a-zA-Z0-9]{5,20}");
    private static final Set<String> FORBINDDDEN_WORD_SET = new HashSet<String>();
    static {
       FORBINDDDEN_WORD_SET.add("fuc k"); //删掉空格
       FORBINDDDEN_WORD_SET.add("admin");
    }
    @Override
    public boolean supports(Class<?> clazz) {
       return UserModel.class == clazz;//表示只对UserModel类型的目标对象实施验证
    }
    @Override
    public void validate(Object target, Errors errors) {
       //这个表示如果目标对象的username属性为空，则表示错误（简化我们手工判断是否为空）
       ValidationUtils.rejectIfEmpty(errors, "username", "username.not.empty");

       UserModel user = (UserModel) target;

       if(!USERNAME_PATTERN.matcher(user.getUsername()).matches()) {
           errors.rejectValue("username", "username.not.illegal");//如果用户名不合法
       }

       for(String forbiddenWord : FORBINDDDEN_WORD_SET) {
           if(user.getUsername().contains(forbiddenWord)) {
              errors.rejectValue("username", "username.forbidden", new Object[]{forbidden
              break;
           }
       }
       if(!PASSWORD_PATTERN.matcher(user.getPassword()).matches()) {
           errors.rejectValue("password","password.not.illegal", "密码不合法");//密码不合法
       }
    }
}
```

supports方法：表示只对UserModel类型的对象验证；

validate方法：数据验证的具体方法，有如下几个验证：

1、用户名不合法（长度5-20，以字母开头，随后可以是字母、数字、下划线）

USERNAME_PATTERN.matcher(user.getUsername()).matches() //使用正则表达式验证

errors.rejectValue("username", "username.not.illegal");//验证失败为username字段添加错误码

2、屏蔽关键词：即用户名中含有不合法的数据（如admin）

user.getUsername().contains(forbiddenWord) //用contains来判断我们的用户名中是否含有非法关键词

errors.rejectValue("username", "username.forbidden", **new** Object[]{forbiddenWord}, "您的用户名包含非法关键词");//验证失败为username字段添加错误码（参数为当前屏蔽关键词）（默认消息为"您的用户名包含非法关键词"）

3、密码不合法

在此就不罗列代码了；

4、ValidationUtils

ValidationUtils.*rejectIfEmpty*(errors, "username", "username.not.empty");

表示如果目标对象的username属性数据为空，则添加它的错误码；

内部通过（value == null || !StringUtils.hasLength(value.toString())）实现判断value是否为空，从而简化代码。

`**（2、spring配置文件chapter4-servlet.xml**`

```
<bean id="userModelValidator"
class="cn.javass.chapter4.web.controller.support.validator.UserModelValidator"/>
<bean name="/validator"
class="cn.javass.chapter4.web.controller.RegisterSimpleFormController">
    <property name="formView" value="registerAndValidator"/>
    <property name="successView" value="redirect:/success"/>
    <property name="validator" ref="userModelValidator"/>
</bean>
```

此处使用了我们第4.9节创建的RegisterSimpleFormController。

（**3**、错误码配置（**messages.properties**），需要执行**NativeToAscii**

```
username.not.empty=用户名不能为空
username.not.illegal=用户名错误，必须以字母开头，只能出现字母、数字、下划线，并且长度在5-20之间
username.forbidden=用户名中包含非法关键词【{0}】
password.not.illegal=密码长度必须在5-20之间
```

（**4**、视图页面（**/WEB-INF/jsp/registerAndValidator.jsp**）

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<form:form commandName="user">

<form:errors path="*" cssStyle="color:red"></form:errors><br/>

username:<form:input path="username"/>
<form:errors path="username" cssStyle="color:red"></form:errors>
<br/>

password:<form:password path="password"/>
<form:errors path="password" cssStyle="color:red"></form:errors>
<br/>
<input type="submit" value="注册"/>
</form:form>
```

form:errors path=*"username"*__：表示只显示username字段的错误信息；

（**5**、测试

地址：http://localhost:9080/springmvc-chapter4/validator

用户名中包含非法关键词【admin】
密码长度必须在5-20之间
username: admin123 用户名中包含非法关键词【admin】
password: 密码长度必须在5-20之间
注册

当我们输入错误的数据后，会报错（form:errors path="*"显示所有错误信息，而form:errors path=*"username"*只显示该字段相关的）。

问题：

如MultiActionController控制器相关方法没有提供给我们errors对象（Errors），我们应该怎么进行错误处理呢？

此处给大家一个思路，errors本质就是一个Errors接口实现，而且在页面要读取相关的错误对象，该错误对象应该存放在模型对象里边，因此我们可以自己创建个errors对象并将其添加到模型对象中即可。

此处我们复制4.15节的UserController类为UserAndValidatorController，并修改它的create（新增）方法添加如下代码片段：

```
BindException errors = new BindException(user, getCommandName(user));
//如果用户名为空
if(!StringUtils.hasLength(user.getUsername())) {
    errors.rejectValue("username", "username.not.empty");
}
if(errors.hasErrors()) {
    return new ModelAndView(getCreateView()).addAllObjects(errors.getModel());
}
```

√ **new BindException(user, getCommandName(user))**：使用当前的命令对象，和命令对象的名字创建了一个BindException作为errors；

√**StringUtils.*hasLength*(user.getUsername())**：如果用户名为空就是用errors.rejectValue("username", "username.not.empty");注入错误码；

√**errors.hasErrors()**：表示如果有错误就返回到新增页面并显示错误消息；

√**ModelAndView(getCreateView()).addAllObjects(errors.getModel())**：此处一定把errors对象的模型数据放在当前的ModelAndView中，作为当前请求的模型数据返回。

在浏览器地址栏输入：http://localhost:9080/springmvc-chapter4/userAndValidator/create 到新增页面

用户名不能为空
用户名：
真实姓名：
新增

用户名什么都不输入，提交后又返回到新增页面 而且显示了错误消息说明我们的想法是正确的。

4.16.2.4.2、声明式验证器

从Spring3开始支持JSR-303验证框架，支持XML风格和注解风格的验证，目前在@RequestMapping时才能使用，也就是说基于Controller接口的实现不能使用该方式（但可以使用编程式验证，有需要的可以参考hibernate validator实现），我们将在第七章详细介绍。

到此Spring2风格的控制器我们就介绍完了，以上控制器从spring3.0开始已经不推荐使用了（但考虑到还有部分公司使用这些@Deprecated类，在此也介绍了一下），而是使用注解控制器实现（@Controller和@RequestMapping）。

第五章 处理器拦截器详解——跟着开涛学SpringMVC

5.1、处理器拦截器简介

`Spring Web MVC的处理器拦截器（如无特殊说明，下文所说的拦截器即处理器拦截器）`

`类似于Servlet开发中的过滤器Filter，用于对处理器进行预处理和后处理。`

5.1.1、常见应用场景

`1、日志记录：``记录请求信息的日志，以便进行信息监控、信息统计、计算PV（Page View）等。`

`2、权限检查：``如登录检测，进入处理器检测检测是否登录，如果没有直接返回到登录页面；`

`3、性能监控：``有时候系统在某段时间莫名其妙的慢，可以通过拦截器在进入处理器之前记录开始时间，在处理完后记录`

`4、通用行为：``读取cookie得到用户信息并将用户对象放入请求，从而方便后续流程使用，还有如提取Locale、Theme`

`5、OpenSessionInView：``如Hibernate，在进入处理器打开Session，在完成后关闭Session。`

`…………本质也是AOP（面向切面编程），也就是说符合横切关注点的所有功能都可以放入拦截器实现。`

5.1.2、拦截器接口

```
package org.springframework.web.servlet;
public interface HandlerInterceptor {
    boolean preHandle(
            HttpServletRequest request, HttpServletResponse response,
            Object handler)
            throws Exception;

    void postHandle(
            HttpServletRequest request, HttpServletResponse response,
            Object handler, ModelAndView modelAndView)
            throws Exception;

    void afterCompletion(
            HttpServletRequest request, HttpServletResponse response,
            Object handler, Exception ex)
            throws Exception;
}
```

`我们可能注意到拦截器一个有3个回调方法，而一般的过滤器Filter才两个，这是怎么回事呢？马上分析。`

preHandle：预处理回调方法，实现处理器的预处理（如登录检查），第三个参数为响应的处理器（如我们上一章的Controller实现）；

返回值：true表示继续流程（如调用下一个拦截器或处理器）；

false表示流程中断（如登录检查失败），不会继续调用其他的拦截器或处理器，此时我们需要通过response来产生响应；

postHandle：后处理回调方法，实现处理器的后处理（但在渲染视图之前），此时我们可以通过modelAndView（模型和视图对象）对模型数据进行处理或对视图进行处理，modelAndView也可能为null。

afterCompletion：整个请求处理完毕回调方法，即在视图渲染完毕时回调，如性能监控中我们可以在此记录结束时间并输出消耗时间，还可以进行一些资源清理，类似于try-catch-finally中的finally，但仅调用处理器执行链中**preHandle**返回**true**的拦截器的**afterCompletion**。

5.1.3、拦截器适配器

有时候我们可能只需要实现三个回调方法中的某一个，如果实现HandlerInterceptor接口的话，三个方法必须实现，不管你需不需要，此时spring提供了一个HandlerInterceptorAdapter适配器（一种适配器设计模式的实现），允许我们只实现需要的回调方法。

```
public abstract class HandlerInterceptorAdapter implements HandlerInterceptor {
     //省略代码 此处所以三个回调方法都是空实现，preHandle返回true。
}
```

5.1.4、运行流程图

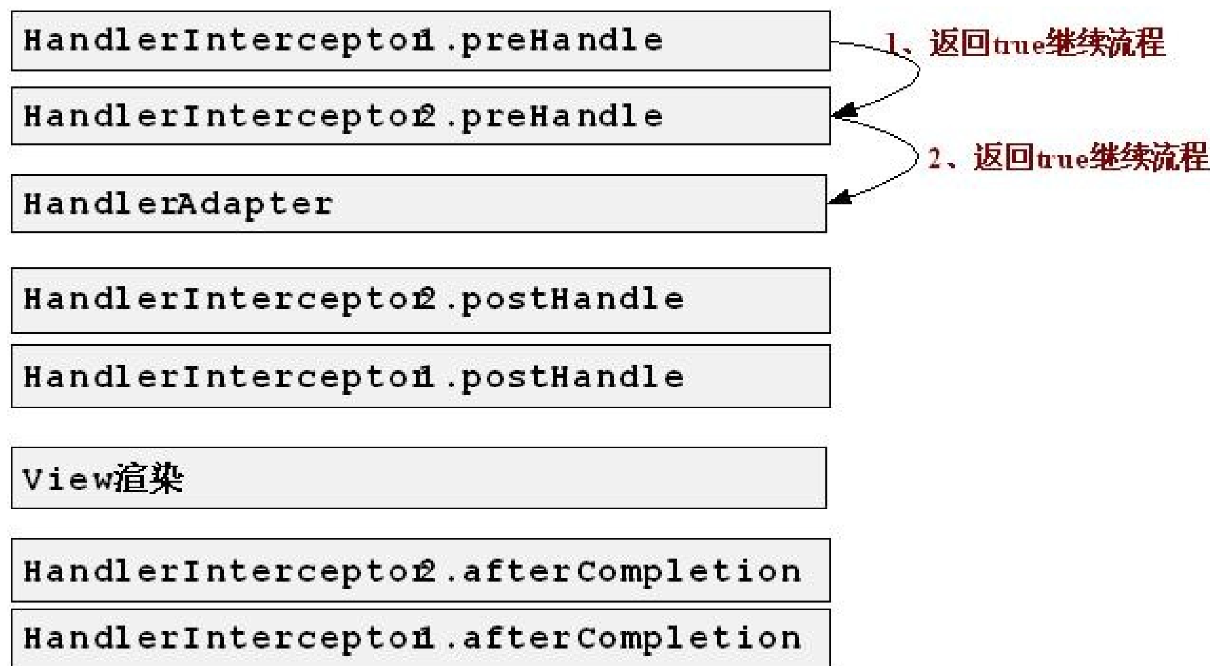


图5-1 正常流程

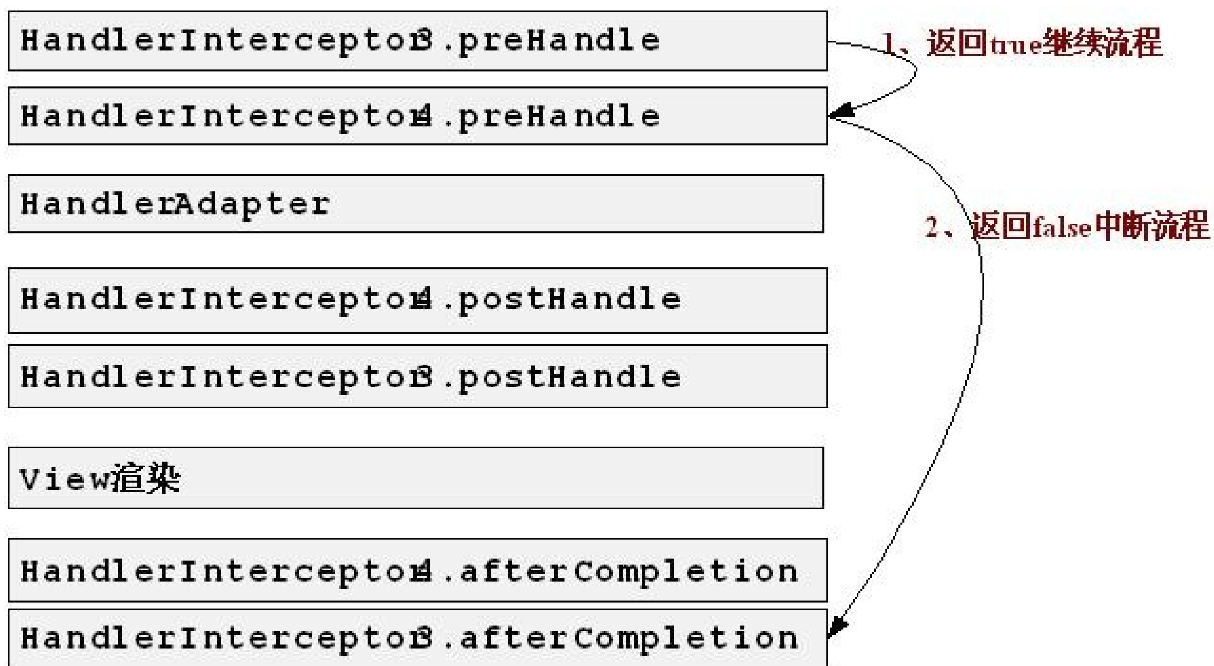


图5-2 中断流程

中断流程中，比如是HandlerInterceptor2中断的流程（preHandle返回false），此处仅调用它之前拦截器的preHandle返回true的afterCompletion方法。

接下来看一下DispatcherServlet内部到底是如何工作的吧：

```
//doDispatch方法
//1、处理器拦截器的预处理（正序执行）
HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();
if (interceptors != null) {
    for (int i = 0; i < interceptors.length; i++) {
    HandlerInterceptor interceptor = interceptors[i];
        if (!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler()
            //1.1、失败时触发afterCompletion的调用
            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, res
            return;
        }
        interceptorIndex = i;//1.2、记录当前预处理成功的索引
}
}
//2、处理器适配器调用我们的处理器
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
//当我们返回null或没有返回逻辑视图名时的默认视图名翻译（详解4.15.5 RequestToViewNameTranslator）
if (mv != null && !mv.hasView()) {
    mv.setViewName(getDefaultViewName(request));
}
//3、处理器拦截器的后处理（逆序）
if (interceptors != null) {
for (int i = interceptors.length - 1; i >= 0; i--) {
      HandlerInterceptor interceptor = interceptors[i];
      interceptor.postHandle(processedRequest, response, mappedHandler.getHandler(), mv);
}
}
//4、视图的渲染
if (mv != null && !mv.wasCleared()) {
render(mv, processedRequest, response);
    if (errorView) {
        WebUtils.clearErrorRequestAttributes(request);
}
//5、触发整个请求处理完毕回调方法afterCompletion
triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, null)
```

注：以上是流程的简化代码，中间省略了部分代码，不完整。

```
// triggerAfterCompletion方法
private void triggerAfterCompletion(HandlerExecutionChain mappedHandler, int interceptorI
            HttpServletRequest request, HttpServletResponse response, Exception ex) throw
        // 5、触发整个请求处理完毕回调方法afterCompletion （逆序从1.2中的预处理成功的索引处的拦截器抄
        if (mappedHandler != null) {
            HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();
            if (interceptors != null) {
                for (int i = interceptorIndex; i >= 0; i--) {
                    HandlerInterceptor interceptor = interceptors[i];
                    try {
                        interceptor.afterCompletion(request, response, mappedHandler.getH
                    }
                    catch (Throwable ex2) {
                        logger.error("HandlerInterceptor.afterCompletion threw exception"
                    }
                }
            }
        }
    }
```

5.2``、入门

具体内容详见工程springmvc-chapter5。

5.2.1、正常流程

（1、拦截器实现

```
package cn.javass.chapter5.web.interceptor;
//省略import
public class HandlerInterceptor1 extends HandlerInterceptorAdapter {//此处一般继承HandlerIn
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Ob
        System.out.println("===========HandlerInterceptor1 preHandle");
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Obje
        System.out.println("===========HandlerInterceptor1 postHandle");
    }
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
        System.out.println("===========HandlerInterceptor1 afterCompletion");
    }
}
```

以上是HandlerInterceptor1实现，HandlerInterceptor2同理 只是输出内容为“HandlerInterceptor2”。

（2、控制器

```
package cn.javass.chapter5.web.controller;
//省略import
public class TestController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) t
        System.out.println("===========TestController");
        return new ModelAndView("test");
    }
}
```

（3、**Spring**配置文件**chapter5-servlet.xml**

```
<bean name="/test" class="cn.javass.chapter5.web.controller.TestController"/>
<bean id="handlerInterceptor1"
class="cn.javass.chapter5.web.interceptor.HandlerInterceptor1"/>
<bean id="handlerInterceptor2"
class="cn.javass.chapter5.web.interceptor.HandlerInterceptor2"/>
```

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors">
        <list>
           <ref bean="handlerInterceptor1"/>
          <ref bean="handlerInterceptor2"/>
        </list>
    </property>
</bean>
```

interceptors：指定拦截器链，拦截器的执行顺序就是此处添加拦截器的顺序；

（**4**、视图页面**WEB-INF/jsp/test.jsp**

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%System.out.println("==========test.jsp");%>
test page
```

在控制台输出 test.jsp

（**5**、启动服务器测试

输入网址：http://localhost:9080/springmvc-chapter5/test

控制台输出：

```
===========HandlerInterceptor1 preHandle
===========HandlerInterceptor2 preHandle
===========TestController
===========HandlerInterceptor2 postHandle
===========HandlerInterceptor1 postHandle
==========test.jsp
===========HandlerInterceptor2 afterCompletion
===========HandlerInterceptor1 afterCompletion
```

到此一个正常流程的演示完毕。和图5-1一样，接下来看一下中断的流程。

5.2.2、中断流程

（**1**、拦截器

HandlerInterceptor3和HandlerInterceptor4 与 之前的 HandlerInteceptor1和 HandlerInterceptor2一样，只是在HandlerInterceptor4的preHandle方法返回false：

```
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Ob
        System.out.println("===========HandlerInterceptor1 preHandle");
response.getWriter().print("break");//流程中断的话需要我们进行响应的处理
        return false;//返回false表示流程中断
}
```

（**2**、控制器

流程中断不会执行到控制器，使用之前的TestController控制器。

（**3**、**Spring**配置文件**chapter5-servlet.xml**

```
<bean id="handlerInterceptor3"
class="cn.javass.chapter5.web.interceptor.HandlerInterceptor3"/>
<bean id="handlerInterceptor4"
class="cn.javass.chapter5.web.interceptor.HandlerInterceptor4"/>
```

```
<bean id="handlerInterceptor3"
class="cn.javass.chapter5.web.interceptor.HandlerInterceptor3"/>
<bean id="handlerInterceptor4"
class="cn.javass.chapter5.web.interceptor.HandlerInterceptor4"/>
```

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors">
        <list>
           <ref bean="handlerInterceptor3"/>
          <ref bean="handlerInterceptor4"/>
        </list>
    </property>
</bean>
```

interceptors：指定拦截器链，拦截器的执行顺序就是此处添加拦截器的顺序；

（**4**、视图页面

流程中断，不会执行到视图渲染。

（**5**、启动服务器测试

输入网址：http://localhost:9080/springmvc-chapter5/test

控制台输出：

```
===========HandlerInterceptor3 preHandle
===========HandlerInterceptor4 preHandle
===========HandlerInterceptor3 afterCompletion
```

此处我们可以看到只有HandlerInterceptor3的afterCompletion执行，否和图5-2的中断流程。

而且页面上会显示我们在HandlerInterceptor4 preHandle 直接写出的响应“break”。

5.3、应用

5.3.1、性能监控

如记录一下请求的处理时间，得到一些慢请求（如处理时间超过500毫秒），从而进行性能改进，一般的反向代理服务器如apache都具有这个功能，但此处我们演示一下使用拦截器怎么实现。

实现分析：

1、在进入处理器之前记录开始时间，即在拦截器的preHandle记录开始时间；

2、在结束请求处理之后记录结束时间，即在拦截器的afterCompletion记录结束实现，并用结束时间-开始时间得到这次请求的处理时间。

问题：

我们的拦截器是单例，因此不管用户请求多少次都只有一个拦截器实现，即线程不安全，那我们应该怎么记录时间呢？

解决方案是使用ThreadLocal，它是线程绑定的变量，提供线程局部变量（一个线程一个ThreadLocal，A线程的ThreadLocal只能看到A线程的ThreadLocal，不能看到B线程的ThreadLocal）。

代码实现：

```
package cn.javass.chapter5.web.interceptor;
public class StopWatchHandlerInterceptor extends HandlerInterceptorAdapter {
    private NamedThreadLocal<Long>  startTimeThreadLocal =
new NamedThreadLocal<Long>("StopWatch-StartTime");
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        long beginTime = System.currentTimeMillis();//1、开始时间
        startTimeThreadLocal.set(beginTime);//线程绑定变量（该数据只有当前请求的线程可见）
        return true;//继续流程
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
        long endTime = System.currentTimeMillis();//2、结束时间
        long beginTime = startTimeThreadLocal.get();//得到线程绑定的局部变量（开始时间）
        long consumeTime = endTime - beginTime;//3、消耗的时间
        if(consumeTime > 500) {//此处认为处理时间超过500毫秒的请求为慢请求
            //TODO 记录到日志文件
            System.out.println(
String.format("%s consume %d millis", request.getRequestURI(), consumeTime));
        }
    }
}
```

NamedThreadLocal：Spring提供的一个命名的ThreadLocal实现。

在测试时需要把stopWatchHandlerInterceptor放在拦截器链的第一个，这样得到的时间才是比较准确的。

5.3.2、登录检测

在访问某些资源时（如订单页面），需要用户登录后才能查看，因此需要进行登录检测。

流程：

1、访问需要登录的资源时，由拦截器重定向到登录页面；

2、如果访问的是登录页面，拦截器不应该拦截；

3、用户登录成功后，往cookie/session添加登录成功的标识（如用户编号）；

4、下次请求时，拦截器通过判断cookie/session中是否有该标识来决定继续流程还是到登录页面；

5、在此拦截器还应该允许游客访问的资源。

拦截器代码如下所示：

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
    //1、请求到登录页面 放行
    if(request.getServletPath().startsWith(loginUrl)) {
        return true;
    }

    //2、TODO 比如退出、首页等页面无需登录，即此处要放行 允许游客的请求

    //3、如果用户已经登录 放行
    if(request.getSession().getAttribute("username") != null) {
        //更好的实现方式的使用cookie
        return true;
    }

    //4、非法请求 即这些请求需要登录后才能访问
    //重定向到登录页面
    response.sendRedirect(request.getContextPath() + loginUrl);
    return false;
}
```

提示：推荐能使用servlet规范中的过滤器Filter实现的功能就用Filter实现，因为HandlerInteceptor只有在Spring Web MVC环境下才能使用，因此Filter是最通用的、最先应该使用的。如登录这种拦截器最好使用Filter来实现。

源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC

源代码请到附件中下载。

其他下载：

跟着开涛学SpringMVC 第一章源代码下载

第二章 Spring MVC入门 源代码下载

Controller接口控制器详解 源代码下载

源码下载——第四章 Controller接口控制器详解——跟着开涛学SpringMVC

源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC

目录：[

第一章 Web MVC简介 —— 跟开涛学SpringMVC](/blog/1593441 "第一章 Web MVC简介 —— 跟开涛学SpringMVC")

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第五章 处理器拦截器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

声明：本系列都是原创内容，觉得好就顶一个，让更多人知道！！ 希望那些踩的人给出不好的理由，我会积极改正。写博客不容易，写原创更不容易！！

6.1、注解式控制器简介

`**一、Spring2.5之前，我们都是通过实现Controller接口或其实现来定义我们的处理器类。已经@Deprecated。**`

`**二、Spring2.5引入注解式处理器支持，通过@Controller 和 @RequestMapping注解定义我们的处理器类。**`

`**并且提供了一组强大的注解：**`

`需要通过处理器映射DefaultAnnotationHandlerMapping和处理器适配器`

`AnnotationMethodHandlerAdapter` 来开启支持@Controller 和

@RequestMapping注解的处理器。

`@Controller：``用于标识是处理器类；`

`@RequestMapping：``请求到处理器功能方法的映射规则；`

`@RequestParam：``请求参数到处理器功能处理方法的方法参数上的绑定；`

`@ModelAttribute：``请求参数到命令对象的绑定；`

`@SessionAttributes：``用于声明session级别存储的属性，放置在处理器类上，通常列出`

`模型属性（如@ModelAttribute）``对应的名称，` 则这些属性会透明的保存到session中；

`@InitBinder：``自定义数据绑定注册支持，用于将请求参数转换到命令对象属性的对应类型；`

`**三、Spring3.0引入RESTful架构风格支持(通过@PathVariable注解和一些其他特性支持),且又引入了**`

`**更多的注解支持：**`

`@CookieValue：``cookie数据到处理器功能处理方法的方法参数上的绑定；`

`@RequestHeader：``请求头（header）数据到处理器功能处理方法的方法参数上的绑定；`

`@RequestBody：``请求的body体的绑定（通过HttpMessageConverter进行类型转换）；`

`@ResponseBody：``处理器功能处理方法的返回值作为响应体（通过HttpMessageConverter进行类型转换）；`

`@ResponseStatus：``定义处理器功能处理方法/异常处理器返回的状态码和原因；`

`@ExceptionHandler：``注解式声明异常处理器；`

`@PathVariable：``请求URI中的模板变量部分到处理器功能处理方法的方法参数上的绑定，`

`从而支持RESTful架构风格的URI；`

四、Spring3.1使用新的HandlerMapping 和 HandlerAdapter来支持@Contoller和@RequestMapping

注解处理器。

新的@Contoller和@RequestMapping注解支持类：处理器映射RequestMappingHandlerMapping

和 处理器适配器RequestMappingHandlerAdapter组合来代替Spring2.5开始的处理器映射DefaultAnnotationHandlerMapping和处理器适配器AnnotationMethodHandlerAdapter，

提供更多的扩展点。

接下来，我们一起开始学习基于注解的控制器吧。

②、④、⑥一般是可变的，因此我们可以这些信息进行请求到处理器的功能处理方法的映射，

因此请求的映射分为如下几种：

URL路径映射：使用URL映射请求到处理器的功能处理方法；

请求方法映射限定：如限定功能处理方法只处理GET请求；

请求参数映射限定：如限定只处理包含“abc”请求参数的请求；

请求头映射限定：如限定只处理“Accept=application/json”的请求。

接下来看看具体如何映射吧。

6.2、入门

（1、控制器实现

```
package cn.javass.chapter6.web.controller;
//省略import
@Controller         // 或 @RequestMapping               //①将一个POJO类声明为处理器
public class HelloWorldController {
    @RequestMapping(value = "/hello")                  //②请求URL到处理器功能处理方法的映射
    public ModelAndView helloWorld() {
        //1、收集参数
        //2、绑定参数到命令对象
        //3、调用业务对象
        //4、选择下一个页面
        ModelAndView mv = new ModelAndView();
        //添加模型数据 可以是任意的POJO对象
        mv.addObject("message", "Hello World!");
        //设置逻辑视图名，视图解析器会根据该名字解析到具体的视图页面
        mv.setViewName("hello");
        return mv;                                         //○3 模型数据和逻辑视图名
    }
}
```

可以通过在一个POJO类上放置@Controller或@RequestMapping，即可把一个POJO类变身为处理器；

@RequestMapping(value = "/hello") ``请求URL(/hello) 到 处理器的功能处理方法的映射；

模型数据和逻辑视图名的返回。

现在的处理器无需实现/继承任何接口/类，只需要在相应的类/方法上放置相应的注解说明下即可，

非常方便。

（2、Spring配置文件chapter6-servlet.xml

（2.1、HandlerMapping和HandlerAdapter的配置

如果您使用的是Spring3.1之前版本，开启注解式处理器支持的配置为：

DefaultAnnotationHandlerMapping 和AnnotationMethodHandlerAdapter。

```
<!—Spring3.1之前的注解 HandlerMapping -->
<bean
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>

<!—Spring3.1之前的注解 HandlerAdapter -->
<bean
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"/>
```

如果您使用的Spring3.1开始的版本，建议使用RequestMappingHandlerMapping 和RequestMappingHandlerAdapter。

```
<!--Spring3.1开始的注解 HandlerMapping -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping
<!--Spring3.1开始的注解 HandlerAdapter -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter
```

下一章我们介绍DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter

与RequestMappingHandlerMapping和RequestMappingHandlerAdapter 的区别。

（2.2、视图解析器的配置

还是使用之前的org.springframework.web.servlet.view.InternalResourceViewResolver。

（2.3、处理器的配置

```
<!-- 处理器 -->
<bean class="cn.javass.chapter6.web.controller.HelloWorldController"/>
```

只需要将处理器实现类注册到spring配置文件即可，spring的DefaultAnnotationHandlerMapping或RequestMapp

能根据注解@Controller或@RequestMapping自动发现。

（2.4、视图页面（/WEB-INF/jsp/hello.jsp）

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Hello World</title>
</head>
<body>
${message}
</body>
</html>
```

${message}：表示显示由HelloWorldController处理器传过来的模型数据。

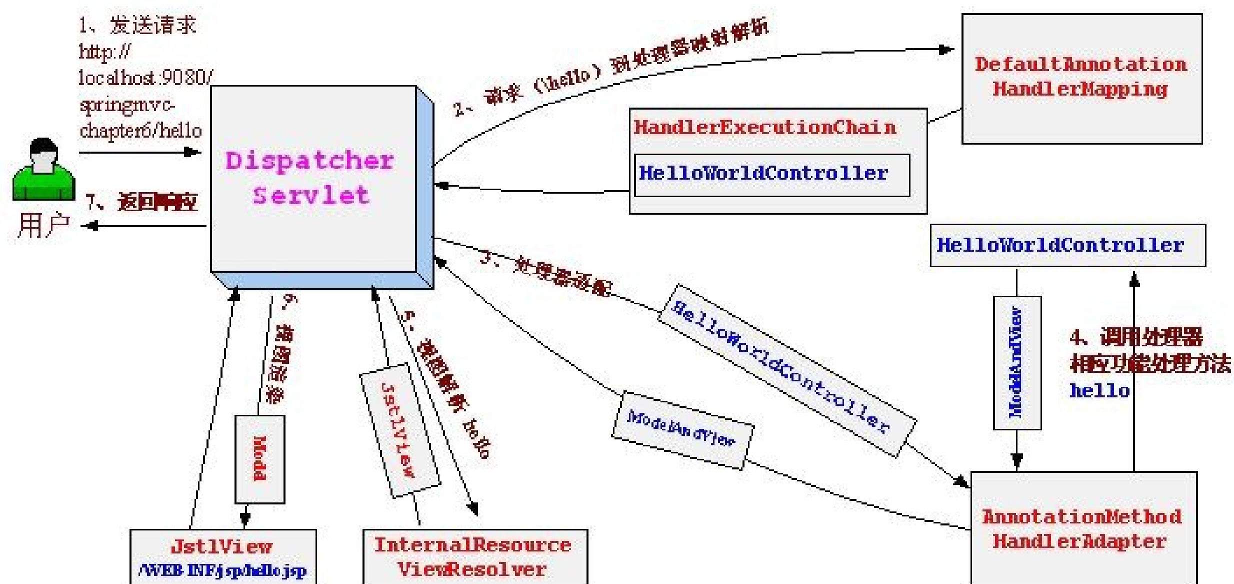
（4、启动服务器测试

地址栏输入http://localhost:9080/springmvc-chapter6/hello，我们将看到页面显示"Hello World!"，

表示成功了。

整个过程和我们第二章中的Hello World 类似，只是处理器的实现不一样。接下来我们来看一下具体流程吧。

6.3、运行流程



和第二章唯一不同的两处是：

1、HandlerMapping实现：使用 `DefaultAnnotationHandlerMapping（spring3.1之前）或RequestMappingHandlerMapping（spring3.1）`

`替换之前的BeanNameUrlHandlerMapping。`

`注解式处理器映射会扫描spring容器中的bean，发现bean实现类上拥有`

`@Controller或@RequestMapping注解的bean，`并将它们作为处理器。

2、HandlerAdapter实现：使用`AnnotationMeth`odHandlerAdapter（spring3.1之前）或RequestMappingHandlerAdapter（spring3.1）替换之前的SimpleControllerHandlerAdapter。

注解式处理器适配器会通过反射调用相应的功能处理方法（方法上拥有`@RequestMapping注解`）。

好了到此我们知道Spring如何发现处理器、如何调用处理的功能处理方法了，接下来我们

详细学习下如何定义处理器、如何进行请求到功能处理方法的定义。

6.4、处理器定义

6.4.1、@Controller

```
@Controller
public class HelloWorldController {
……
}
```

推荐使用这种方式声明处理器，它和我们的@Service、@Repository很好的对应了我们常见的三层开发架构的组件。

6.4.2、@RequestMapping

```
@RequestMapping
public class HelloWorldController {
……
}
```

这种方式也是可以工作的，但如果在类上使用@ RequestMapping注解一般是用于

窄化功能处理方法的映射的，详见6.4.3。

```
package cn.javass.chapter6.web.controller;
@Controller
@RequestMapping(value="/user")                 //①处理器的通用映射前缀
public class HelloWorldController2 {
    @RequestMapping(value = "/hello2")        //②相对于①处的映射进行窄化
    public ModelAndView helloWorld() {
         //省略实现
    }
}
```

6.4.3、窄化请求映射

```
package cn.javass.chapter6.web.controller;
@Controller
@RequestMapping(value="/user")                    //①处理器的通用映射前缀
public class HelloWorldController2 {
    @RequestMapping(value = "/hello2")        //②相对于①处的映射进行窄化
    public ModelAndView helloWorld() {
         //省略实现
    }
}
```

①类上的@RequestMapping(value="/user") 表示处理器的通用请求前缀；

②处理器功能处理方法上的是对①处映射的窄化。

因此http://localhost:9080/springmvc-chapter6/hello2 无法映射到HelloWorldController2的helloWorld功能处理方法；而http://localhost:9080/springmvc-chapter6/user/hello2是可以的。

```
http://localhost:9080/springmvc-chapter6 /user  /hello2
                                  ①         ②
@Controller
@RequestMapping(value="/user")              //①处理器的通用映射前缀
public class HelloWorldController2
    @RequestMapping(value = "/hello2")      //②相对于①处的映射进行窄化
    public ModelAndView helloWorld() {
        //省略实现
    }
}
```

窄化请求映射可以认为是方法级别的@RequestMapping继承类级别的@RequestMapping。

窄化请求映射还有其他方式，如在类级别指定URL，而方法级别指定请求方法类型或参数等等，

后续会详细介绍。

到此，我们知道如何定义处理器了，接下来我们需要学习如何把请求映射到相应的功能处理方法

进行请求处理。

6.5、请求映射

处理器定义好了，那接下来我们应该定义功能处理方法，接收用户请求处理并选择视图进行渲染。

首先我们看一下图6-1:




```
http请求信息包含六部分信息：
```

```
①请求方法，如GET或POST，表示提交的方式；
```

```
②URL，请求的地址信息；
```

```
③协议及版本；
```

```
④请求头信息（包括Cookie信息）；
```

```
⑤回车换行（CRLF）；
```

```
⑥请求内容区（即请求的内容或数据），如表单提交时的参数数据、URL请求参数（?abc=123 ？后边的）等。
```

```
想要了解HTTP/1.1协议，请访问[http://tools.ietf.org/html/rfc2616](http://tools.ietf.org/html/
```

```
那此处我们可以看到有①、②、④、⑥一般是可变的，因此我们可以这些信息进行请求到
```

```
处理器的功能处理方法的映射，因此请求的映射分为如下几种：
```

URL路径映射：使用URL映射请求到处理器的功能处理方法；

请求方法映射限定：如限定功能处理方法只处理GET请求；

请求参数映射限定：如限定只处理包含“abc”请求参数的请求；

请求头映射限定：如限定只处理“Accept=application/json”的请求。

接下来看看具体如何映射吧。

源代码下载 第六章 注解式控制器详解

源代码请到附件中下载。

其他下载：

跟着开涛学SpringMVC 第一章源代码下载

第二章 Spring MVC入门 源代码下载

Controller接口控制器详解 源代码下载

源码下载——第四章 Controller接口控制器详解——跟着开涛学SpringMVC

源代码下载 第五章 处理器拦截器详解——跟着开涛学SpringMVC

目录：[

第一章 Web MVC简介 —— 跟开涛学SpringMVC](/blog/1593441 "第一章 Web MVC简介 —— 跟开涛学SpringMVC")

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第五章 处理器拦截器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

SpringMVC3强大的请求映射规则详解 第六章 注解式控制器详解——跟着开涛学SpringMVC

声明：本系列都是原创内容，觉得好就顶一个，让更多人知道！！写博客不容易，写原创更不容易！！

6.5、请求映射

处理器定义好了，那接下来我们应该定义功能处理方法，接收用户请求处理并选择视图进行渲染。首先我们看一下图6-1:

```
http请求信息包含六部分信息：
```

```
①请求方法，如GET或POST，表示提交的方式；
```

```
②URL，请求的地址信息；
```

```
③协议及版本；
```

```
④请求头信息（包括Cookie信息）；
```

```
⑤回车换行（CRLF）；
```

```
⑥请求内容区（即请求的内容或数据），如表单提交时的参数数据、URL请求参数（?abc=123 ？后边的）等。
```

```
想要了解HTTP/1.1协议，请访问[http://tools.ietf.org/html/rfc2616](http://tools.ietf.org/html/
```

```
那此处我们可以看到有①、②、④、⑥一般是可变的，因此我们可以这些信息进行请求到处理器的功能处理方法的映射，
```

URL路径映射：使用URL映射请求到处理器的功能处理方法；

请求方法映射限定：如限定功能处理方法只处理GET请求；

请求参数映射限定：如限定只处理包含“abc”请求参数的请求；

请求头映射限定：如限定只处理“Accept=application/json”的请求。

接下来看看具体如何映射吧。

6.5.1、URL路径映射

6.5.1.1、普通URL路径映射

@RequestMapping(value={"/test1", "/user/create"})：多个URL路径可以映射到同一个处理器的功能处理方法。

6.5.1.2、URI模板模式映射

@RequestMapping(value="/users/{userId}")：{×××}占位符， 请求的URL可以是“/users/123456”或

“/users/abcd”，通过6.6.5讲的通过@PathVariable可以提取URI模板模式中的{×××}中的×××变量。

@RequestMapping(value="/users/{userId}/create")：这样也是可以的，请求的URL可以是“/users/123/create”。

@RequestMapping(value="/users/{userId}/topics/{topicId}")：这样也是可以的，请求的URL可以是“/users/123/topics/123”。

6.5.1.3、Ant风格的URL路径映射

@RequestMapping(value="/users/**")：可以匹配**“/users/abc/abc”**，但**“/users/123”**将会被【**URI**模板模式映射中的**“/users/{userId}”**模式优先映射到】【详见4.14的最长匹配优先**】。

@RequestMapping(value="/product?")：可匹配“/product1”或“/producta”，但不匹配“/product”或“/productaa”;

@RequestMapping(value="/product*")：可匹配“/productabc”或“/product”，但不匹配“/productabc/abc”;

@RequestMapping(value="/product/*")：可匹配“/product/abc”，但不匹配“/productabc”;

@RequestMapping(value="/products/**/{productId}")：可匹配“/products/abc/abc/123”或“/products/123”，也就是Ant风格和URI模板变量风格可混用;

此处需要注意的是【**4.14**中提到的最长匹配优先】，Ant风格的模式请参考4.14。

6.5.1.4、正则表达式风格的URL路径映射

从Spring3.0开始支持正则表达式风格的URL路径映射，格式为{变量名:正则表达式}，这样我们就可以通过6.6.5讲的通过@PathVariable提取模式中的{×××：正则表达式匹配的值}中的×××变量了。

@RequestMapping(value="/products/{categoryCode:\d+}-{pageNumber:\d+}")：可以匹配“/products/123-1”，但不能匹配“/products/abc-1”，这样可以设计更加严格的规则。

正则表达式风格的**URL**路径映射是一种特殊的**URI**模板模式映射：

URI模板模式映射是{userId}，不能指定模板变量的数据类型，如是数字还是字符串；

正则表达式风格的URL路径映射，可以指定模板变量的数据类型，可以将规则写的相当复杂。

6.5.1.5、组合使用是“或”的关系

如 @RequestMapping(value={"/test1", "/user/create"})组合使用是或的关系，即“/test1”或“/user/create”请求URL路径都可以映射到@RequestMapping指定的功能处理方法。

以上URL映射的测试类为：cn.javass.chapter6.web.controller.mapping.MappingController.java。

到此，我们学习了Spring Web MVC提供的强大的URL路径映射，而且可以实现非常复杂的URL规则。Spring Web MVC不仅仅提供URL路径映射，还提供了其他强大的映射规则。接下来我们看一下请求方法映射限定吧。

6.5.2、请求方法映射限定

一般我们熟悉的表单一般分为两步：第一步展示，第二步提交，如4.9、SimpleFormController那样，那如何通过@RequestMapping来实现呢？

6.5.2.1、请求方法映射限定

我们熟知的，展示表单一般为GET请求方法；提交表单一般为POST请求方法。但6.5.1节讲的URL路径映射方式对任意请求方法是全盘接受的，因此我们需要某种方式来告诉相应的功能处理方法只处理如GET请求方法的请求或POST请求方法的请求。

接下来我们使用@RequestMapping来实现SimpleFormController的功能吧。

```
package cn.javass.chapter6.web.controller.method;
//省略import
@Controller
@RequestMapping("/customers/**")                                        //①处理器的通用映射前缀
public class RequestMethodController {
    @RequestMapping(value="/create", method = RequestMethod.GET)//②类级别的@RequestMapping
    public String showForm() {
        System.out.println("===============GET");
        return "customer/create";
    }
    @RequestMapping(value="/create", method = RequestMethod.POST)//③类级别的@RequestMappin
    public String submit() {
        System.out.println("================POST");
        return "redirect:/success";
    }
}
```

①处理器的通用映射前缀（父路径）：表示该处理器只处理匹配“/customers/**”的请求；

②对类级别的@RequestMapping进行窄化，表示showForm可处理匹配“/customers/**/create”且请求方法为“GET”的请求；

③对类级别的@RequestMapping进行窄化，表示submit可处理匹配“/customers/**/create”且请求方法为“POST”的请求。

6.5.2.2、组合使用是“或”的关系

@RequestMapping(value="/methodOr", method = {RequestMethod.*POST*, RequestMethod.*GET*})：即请求方法可以是 GET 或 POST。

提示：

1、一般浏览器只支持GET、POST请求方法，如想浏览器支持PUT、DELETE等请求方法只能模拟，稍候章节介绍。

2、除了GET、POST，还有HEAD、OPTIONS、PUT、DELETE、TRACE。

3、DispatcherServlet默认开启对 GET、POST、PUT、DELETE、HEAD的支持；

4、如果需要支持OPTIONS、TRACE，请添加DispatcherServlet在web.xml的初始化参数：dispatchOptionsRequest 和 dispatchTraceRequest 为true。

请求方法的详细使用请参考RESTful架构风格一章。

以上请求方法映射限定测试类为：cn.javass.chapter6.web.controller.method.RequestMethodController。

6.5.3、请求参数数据映射限定

6.5.3.1、请求数据中有指定参数名

```
package cn.javass.chapter6.web.controller.parameter;
//省略import
@Controller
@RequestMapping("/parameter1")                                    //①处理器的通用映射前缀
public class RequestParameterController1 {
    //②进行类级别的@RequestMapping窄化
    @RequestMapping(params="create", method=RequestMethod.GET)
    public String showForm() {
        System.out.println("==============showForm");
        return "parameter/create";
    }
    //③进行类级别的@RequestMapping窄化
    @RequestMapping(params="create", method=RequestMethod.POST)
    public String submit() {
        System.out.println("================submit");
        return "redirect:/success";
    }
}
```

②@RequestMapping(params="create", method=RequestMethod.*GET*) ：表示请求中有"create"的参数名且请求方法为"GET"即可匹配，如可匹配的请求URL"http://×××/parameter1?create"；

③@RequestMapping(params="create", method=RequestMethod.*POST*)：表示请求中有"create"的参数名且请求方法为"POST"即可匹配；

此处的create请求参数名表示你请求的动作，即你想要的功能的一个标识，常见的CRUD(增删改查)我们可以使用如下请求参数名来表达：

◇（create请求参数名 且 GET请求方法） 新增页面展示、（create请求参数名 且 POST请求方法）新增提交；

◇（update请求参数名 且 GET请求方法） 新增页面展示、（update请求参数名 且 POST请求方法）新增提交；

◇（delete请求参数名 且 GET请求方法） 新增页面展示、（delete请求参数名 且 POST请求方法）新增提交；

◇（query请求参数名 且 GET请求方法） 新增页面展示、（query请求参数名 且 POST请求方法） 新增提交；

◇（list请求参数名 且 GET请求方法） 列表页面展示；

◇（view请求参数名 且 GET请求方法） 查看单条记录页面展示。

6.5.3.2、请求数据中没有指定参数名

```
//请求参数不包含 create参数名
@RequestMapping(params="!create", method=RequestMethod.GET)//进行类级别的@RequestMapping窄化
```

@RequestMapping(params="!create", method=RequestMethod.*GET*)：表示请求中没有“create”参数名且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?abc”。

6.5.3.3、请求数据中指定参数名=值

```
package cn.javass.chapter6.web.controller.parameter;
//省略import
@Controller
@RequestMapping("/parameter2")                      //①处理器的通用映射前缀
public class RequestParameterController2 {
    //②进行类级别的@RequestMapping窄化
    @RequestMapping(params="submitFlag=create", method=RequestMethod.GET)
    public String showForm() {
        System.out.println("==============showForm");
        return "parameter/create";
    }
    //③进行类级别的@RequestMapping窄化
    @RequestMapping(params="submitFlag=create", method=RequestMethod.POST)
    public String submit() {
        System.out.println("==============submit");
        return "redirect:/success";
    }
}
```

②@RequestMapping(params="submitFlag=create", method=RequestMethod.*GET*)：表示请求中有“submitFlag=create”请求参数且请求方法为“GET”即可匹配，如请求URL为http://×××/parameter2?submitFlag=create；

③@RequestMapping(params="submitFlag=create", method=RequestMethod.*POST*)：表示请求中有“submitFlag=create”请求参数且请求方法为“POST”即可匹配；

此处的submitFlag=create请求参数表示你请求的动作，即你想要的功能的一个标识，常见的CRUD(增删改查)我们可以使用如下请求参数名来表达：

◇（submitFlag=create请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=create请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=update请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=update请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=delete请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=delete请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=query请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=query请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=list请求参数名 且 GET请求方法） 列表页面展示；

◇（submitFlag=view请求参数名 且 GET请求方法） 查看单条记录页面展示。

6.5.3.4、请求数据中指定参数名!=值

```
//请求参数submitFlag 不等于 create
@RequestMapping(params="submitFlag!=create", method=RequestMethod.GET)
```

@RequestMapping(params="submitFlag!=create", method=RequestMethod.*GET*)：表示请求中的参数“submitFlag!=create”且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?submitFlag=abc”。

6.5.3.5、组合使用是“且”的关系

```
@RequestMapping(params={"test1", "test2=create"})  //②进行类级别的@RequestMapping窄化
```

@RequestMapping(params={"test1", "test2=create"})：表示请求中的有“test1”参数名 且有“test2=create”参数即可匹配，如可匹配的请求URL“http://×××/parameter3?test1&test2=create。

以上请求参数数据映射限定测试类为：cn.javass.chapter6.web.controller.method包下的RequestParameterController1、RequestParameterController2、RequestParameterController3。

6.5.4、请求头数据映射限定

6.5.4.1、准备环境

浏览器：建议chrome最新版本；

插件：ModHeader

安装地址：https://chrome.google.com/webstore/detail/idgpnmonknjnojddfkpgkljpfnnfcklj

插件安装步骤：

1、打开https://chrome.google.com/webstore/detail/idgpnmonknjnojddfkpgkljpfnnfcklj，如图6-2



图6-2

2、点击“添加至chrome”后弹出“确认安装”对话框，点击“安装”按钮即可，如图6-3：



图6-3

3、安装成功后，在浏览器右上角出现如图6-4的图标表示安装成功：

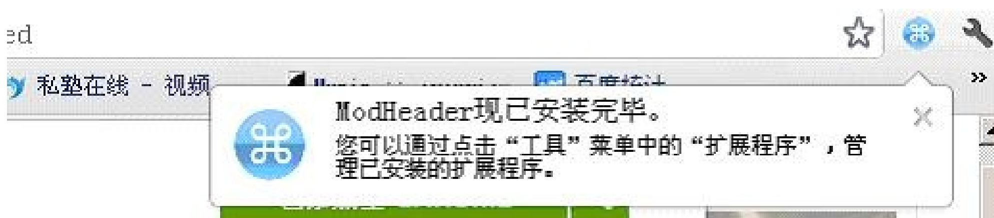


图6-4

4、鼠标右击右上角的“Modify Header”图标，选择选项，打开如图6-5：



图6-5

7、修改完成后，输入URL请求，你可以在chrome的“开发人员工具的”网络选项卡下，看到如图6-7的信息表示添加请求头成功了：



图6-7

到此我们的工具安装完毕，接下来看看如何使用请求头数据进行映射限定。

6.5.4.2、请求头数据中有指定参数名

@RequestMapping(value="/header/test1", headers = "Accept")：表示请求的URL必须为“/header/test1”

且 请求头中必须有Accept参数才能匹配。

@RequestMapping(value="/header/test1", headers = "abc")：表示请求的URL必须为“/header/test1”

且 请求头中必须有abc参数才能匹配，如图6-8时可匹配。

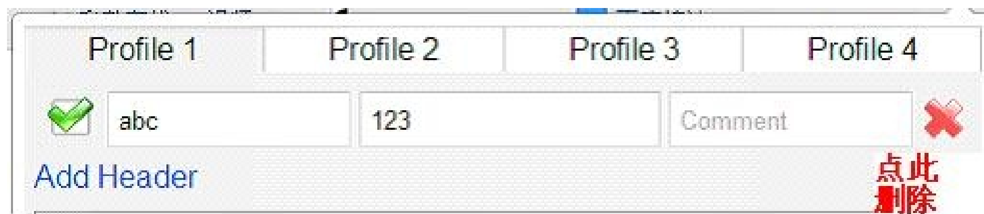


图6-8

6.5.4.3、请求头数据中没有指定参数名

@RequestMapping(value="/header/test2", headers = "!abc")：表示请求的URL必须为“/header/test2”

且 请求头中必须没有abc参数才能匹配。（将Modify Header的abc参数值删除即可）。



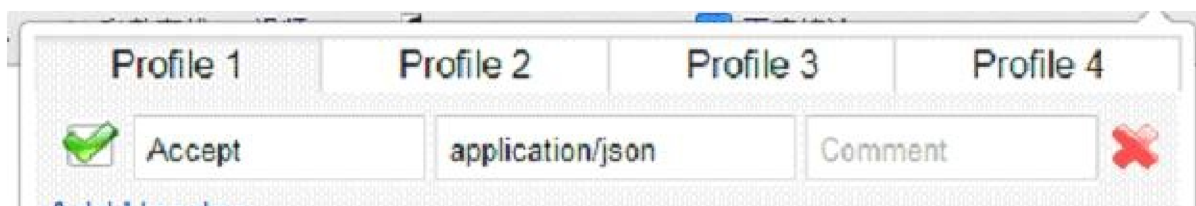
6.5.4.4、请求头数据中指定参数名=值

@RequestMapping(value="/header/test3", headers = "Content-Type=application/json")：表示请求的URL必须为“/header/test3” 且 请求头中必须有“Content-Type=application/json”参数即可匹配。（将Modify Header的Content-Type参数值改为“application/json”即可）；



当你请求的URL为“/header/test3” 但 如果请求头中没有或不是“Content-Type=application/json”参数（如“text/html”其他参数），将返回“HTTP Status 415”状态码【表示不支持的媒体类型(Media Type)，也就是MIME类型】，即我们的功能处理方法只能处理application/json的媒体类型。

@RequestMapping(value="/header/test4", headers = "Accept=application/json")：表示请求的URL必须为“/header/test4” 且 请求头中必须有“Accept =application/json”参数即可匹配。（将Modify Header的Accept参数值改为“application/json”即可）；



当你请求的URL为“/header/test4” 但 如果请求头中没有“Accept=application/json”参数（如“text/html”其他参数），将返回“HTTP Status 406”状态码【不可接受，服务器无法根据Accept头的媒体类型为客户端生成响应】，即客户只接受“application/json”媒体类型的数据，即我们的功能处理方法的响应只能返回“application/json”媒体类型的数据。

@RequestMapping(value="/header/test5", headers = "Accept=text/*") ：表示请求的URL必须为“/header/test5” 且 请求头中必须有如“Accept=text/plain”参数即可匹配。（将Modify Header的Accept参数值改为“text/plain”即可）；

Accept=text/*：表示主类型为text，子类型任意，如“text/plain”、“text/html”等都可以匹配。

@RequestMapping(value="/header/test6", headers = "Accept=/") ：表示请求的URL必须为“/header/test6” 且 请求头中必须有任意Accept参数即可匹配。（将Modify Header的Accept参数值改为“text/html”或“application/xml”等都可以）。

Accept=/：表示主类型任意，子类型任意，如“text/plain”、“application/xml”等都可以匹配。

6.5.4.5、请求头数据中指定参数名!=值

@RequestMapping(value="/header/test7", headers = "Accept!=text/vnd.wap.wml")：表示请求的URL必须为“/header/test7” 且 请求头中必须有“Accept”参数但值不等于“text/vnd.wap.wml”即可匹配。

6.5.4.6、组合使用是“且”的关系

@RequestMapping(value="/header/test8", headers = {"Accept!=text/vnd.wap.wml","abc=123"})：表示请求的URL必须为“/header/test8” 且 请求头中必须有“Accept”参数但值不等于“text/vnd.wap.wml”且 请求中必须有参数“abc=123”即可匹配。

注：Accept:text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8

如果您的请求中含有Accept：“/”，则可以匹配功能处理方法上的如“text/html”、“text/*”，“application/xml”等。

源代码下载

目录：[

第一章 Web MVC简介 —— 跟开涛学SpringMVC](/blog/1593441 "第一章 Web MVC简介 —— 跟开涛学SpringMVC")

第二章 Spring MVC入门 —— 跟开涛学SpringMVC

第三章 DispatcherServlet详解 ——跟开涛学SpringMVC

第四章 Controller接口控制器详解（1）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（2）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（3）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（4）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（5）——跟着开涛学SpringMVC

第四章 Controller接口控制器详解（6）——跟着开涛学SpringMVC

第五章 处理器拦截器详解——跟着开涛学SpringMVC

注解式控制器运行流程及处理器定义 第六章 注解式控制器详解——跟着开涛学SpringMVC

Spring MVC 3.1新特性 生产者、消费者请求限定 —— 第六章 注解式控制器详解——跟着开涛学SpringMVC

6.6.5、生产者、消费者限定

6.6.5.1、基本概念

首先让我们看一下通过HTTP协议传输的媒体类型及如何表示媒体类型：

一、Media Type：

互联网媒体类型，一般就是我们所说的MIME类型，用来确定请求的内容类型或响应的内容类型。

写道媒体类型格式：type/subtype(;parameter)? type主类型，任意的字符串，如text，如果是号代表所有；*subtype* 子类型，任意的字符串，如*html*，如果是号代表所有； parameter 可选，一些参数，如Accept请求头的q参数， Content-Type的 charset参数。

详见http://tools.ietf.org/html/rfc2616#section-3.7

常见媒体类型：

text/html ： HTML格式 text/plain ：纯文本格式 text/xml ：XML格式

image/gif ：gif图片格式 image/jpeg ：jpg图片格式 image/png：png图片格式

application/x-www-form-urlencoded ： <form encType="">中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式）。

multipart/form-data ： 当你需要在表单中进行文件上传时，就需要使用该格式；

application/xhtml+xml ：XHTML格式 application/xml ： XML数据格式

application/atom+xml ：Atom XML聚合格式 application/json ： JSON数据格式

application/pdf ：pdf格式 application/msword ： Word文档格式

application/octet-stream ： 二进制流数据（如常见的文件下载）。

在如tomcat服务器的 "conf/web.xml"中指定了扩展名到媒体类型的映射，在此我们可以看到服务器支持的媒体类型。

二、Content-Type：内容类型，即请求/响应的内容区数据的媒体类型；

2.1、请求头的内容类型，表示发送到服务器的内容数据的媒体类型；

request中设置请求头“Content-Type: application/x-www-form-urlencoded”表示请求的数据为key/value数据；

（1、控制器 **cn.javass.chapter6.web.controller.consumesproduces.contenttype.RequestContentTypeController**

```
    @RequestMapping(value = "/ContentType", method = RequestMethod.GET)
    public String showForm() throws IOException {
        //form表单，使用application/x-www-form-urlencoded编码方式提交表单
        return "consumesproduces/Content-Type";
    }

    @RequestMapping(value = "/ContentType", method = RequestMethod.POST,
headers = "Content-Type=application/x-www-form-urlencoded")
    public String request1(HttpServletRequest request) throws IOException {
        //①得到请求的内容区数据的类型
        String contentType = request.getContentType();
        System.out.println("========ContentType:" + contentType);
        //②得到请求的内容区数据的编码方式，如果请求中没有指定则为null
        //注意，我们的CharacterEncodingFilter这个过滤器设置了编码(UTF-8)
        //编码只能被指定一次，即如果客户端设置了编码，则过滤器不会再设置
        String characterEncoding = request.getCharacterEncoding();
        System.out.println("========CharacterEncoding:" + characterEncoding);

        //③表示请求的内容区数据为form表单提交的参数，此时我们可以通过request.getParameter得到数据（
        System.out.println(request.getParameter("realname"));
        System.out.println(request.getParameter("username"));
        return "success";
    }
```

showForm功能处理方式：展示表单，且form的enctype="application/x-www-form-urlencoded"，在提交时请求的内容类型头为“Content-Type:application/x-www-form-urlencoded”；

request1功能处理方法：只对请求头为“Content-Type:application/x-www-form-urlencoded”的请求进行处理（即消费请求内容区数据）；

request.getContentType()：可以得到请求头的内容区数据类型（即Content-Type头的值）

request.getCharacterEncoding()：如“Content-Type:application/json;charset=GBK”,则得到的编码为“GBK”，否则如果你设置过滤器（CharacterEncodingFilter）则得到它设置的编码，否则返回null。

request.getParameter()：因为请求的内容区数据为application/x-www-form-urlencoded格式的数据，因此我们可以通过request.getParameter()得到相应参数数据。

request中设置请求头“Content-Type:application/json;charset=GBK”表示请求的内容区数据为json类型数据，且内容区的数据以GBK进行编码；

（1、控制器 **cn.javass.chapter6.web.controller.consumesproduces.contenttype.RequestContentTypeController**

```
@RequestMapping(value = "/request/ContentType", method = RequestMethod.POST,
headers = "Content-Type=application/json")
    public String request2(HttpServletRequest request) throws IOException {
        //①表示请求的内容区数据为json数据
        InputStream is = request.getInputStream();
        byte bytes[] = new byte[request.getContentLength()];
        is.read(bytes);
        //②得到请求中的内容区数据（以CharacterEncoding解码）
        //此处得到数据后你可以通过如json-lib转换为其他对象
        String jsonStr = new String(bytes, request.getCharacterEncoding());
        System.out.println("json data：" + jsonStr);
        return "success";
    }
```

request2功能处理方法：只对请求头为“Content-Type:application/json”的进行请求处理（即消费请求内容区数据）；

request.getContentLength()：可以得到请求头的内容区数据的长度；

request.getCharacterEncoding()：如“Content-Type:application/json;charset=GBK”,则得到的编码为“GBK”，否则如果你设置过滤器（CharacterEncodingFilter）则得到它设置的编码，否则返回null。

我们得到json的字符串形式后就能很简单的转换为JSON相关的对象。

（**2**、客户端发送**json**数据请求

```
        //请求的地址
        String url = "http://localhost:9080/springmvc-chapter6/request/ContentType";
        //①创建Http Request(内部使用HttpURLConnection)
        ClientHttpRequest request =
            new SimpleClientHttpRequestFactory().
                createRequest(new URI(url), HttpMethod.POST);
        //②设置请求头的内容类型头和内容编码（GBK）
        request.getHeaders().set("Content-Type", "application/json;charset=gbk");
        //③以GBK编码写出请求内容体
        String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";
        request.getBody().write(jsonData.getBytes("gbk"));
        //④发送请求并得到响应
        ClientHttpResponse response = request.execute();
        System.out.println(response.getStatusCode());
```

此处我们使用Spring提供的Http客户端API SimpleClientHttpRequestFactory创建了请求并设置了请求的Content-Type和编码并在响应体中写回了json数据（即生产json类型的数据），此处是硬编码，实际工作可以使用json-lib等工具进行转换。

具体代码在cn.javass.chapter6.web.controller.consumesproduces.contenttype.RequestContentTypeClient。

2.2、响应头的内容类型，表示发送到客户端的内容数据类型，和请求头的内容类型类似，只是方向相反。

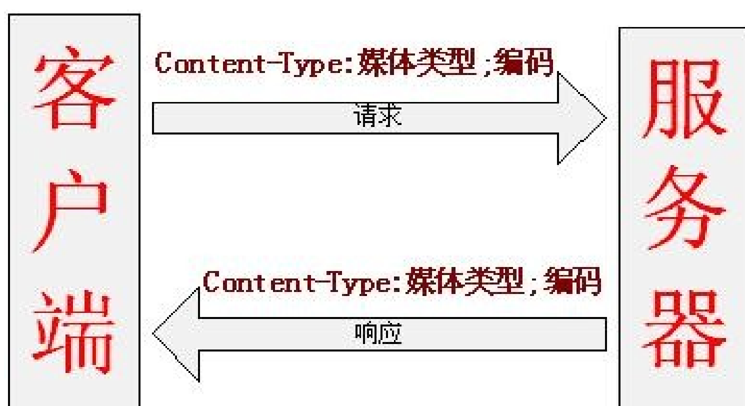
```
@RequestMapping("/response/ContentType")
public void response1(HttpServletResponse response) throws IOException {
    //①表示响应的内容区数据的媒体类型为html格式，且编码为utf-8(客户端应该以utf-8解码)
    response.setContentType("text/html;charset=utf-8");
    //②写出响应体内容
    response.getWriter().write("<font style='color:red'>hello</font>");
}
```

<!--[endif]-->

如上所示，通过response.setContentType("text/html;charset=utf-8") 告诉客户端响应体媒体类型为html，编码为utf-8，大家可以通过chrome工具查看响应头为“Content-Type:text/html;charset=utf-8”，还一个“Content-Length:36”表示响应体大小。

代码在 cn.javass.chapter6.web.controller.consumesproduces.contenttype.ResponseContentTypeController。

如上代码可以看出**Content-Type**可以指定请求**/**响应的内容体的媒体格式和可选的编码方式。如图**6-9**



①客户端—发送请求—服务器：客户端通过请求头Content-Type指定内容体的媒体类型（即客户端此时是生产者），服务器根据Content-Type消费内容体数据（即服务器此时是消费者）；

②服务器—发送请求—客户端：服务器生产响应头Content-Type指定的响应体数据（即服务器此时是生产者），客户端根据Content-Type消费内容体数据（即客户端此时是消费者）。

问题：

①服务器端可以通过指定【headers = "Content-Type=application/json"】来声明可处理（可消费）的媒体类型，即只消费Content-Type指定的请求内容体数据；

②客户端如何告诉服务器端它只消费什么媒体类型的数据呢？即客户端接受（需要）什么类型的数据呢？服务器应该生产什么类型的数据？此时我们可以请求的Accept请求头来实现这个功能。

三、**Accept**：用来指定什么媒体类型的响应是可接受的，即告诉服务器我需要什么媒体类型的数据，此时服务器应该根据Accept请求头生产指定媒体类型的数据。

2.1、json数据

（**1**、服务器端控制器

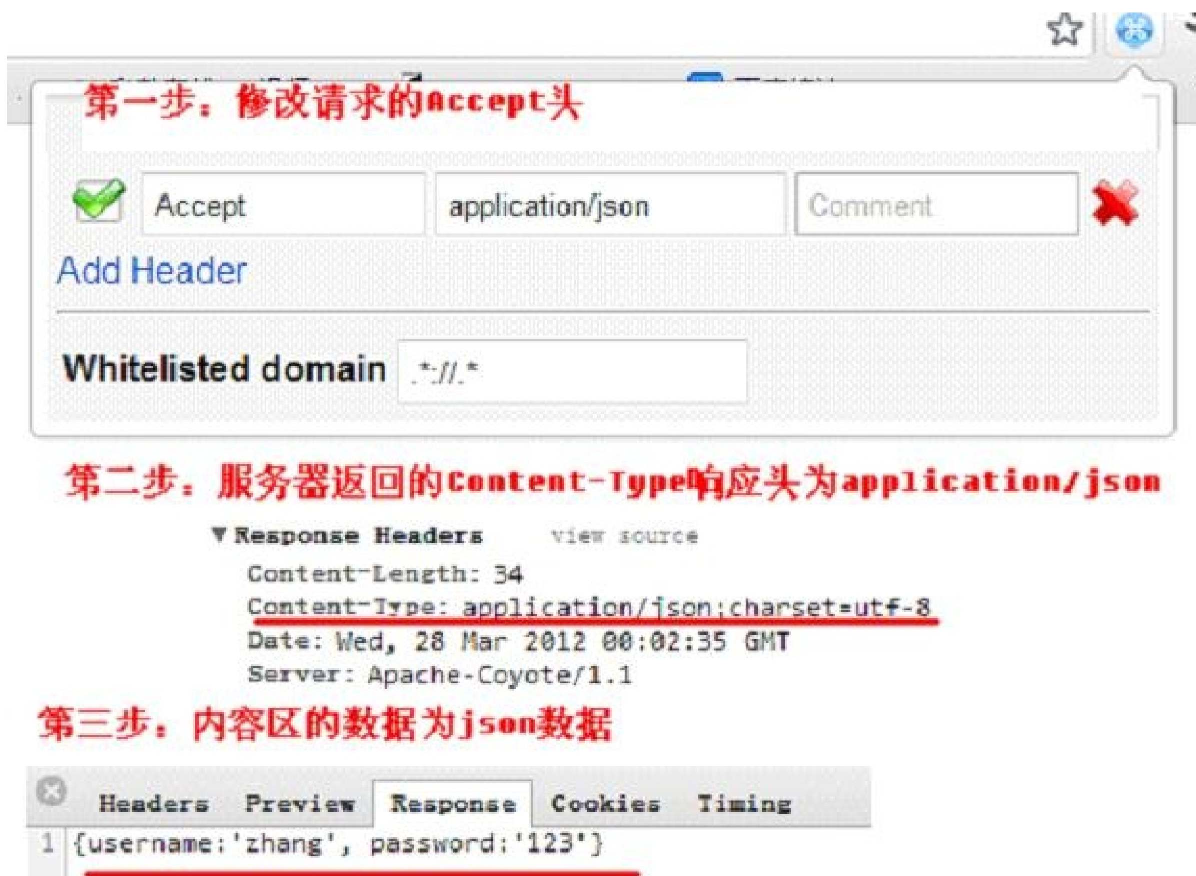
```
    @RequestMapping(value = "/response/ContentType", headers = "Accept=application/json")
    public void response2(HttpServletResponse response) throws IOException {
        //①表示响应的内容区数据的媒体类型为json格式，且编码为utf-8(客户端应该以utf-8解码)
        response.setContentType("application/json;charset=utf-8");
        //②写出响应体内容
        String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";
        response.getWriter().write(jsonData);
    }
```

服务器根据请求头“Accept=application/json”生产json数据。

（**2**、客户端端接收服务器端**json**数据响应

使用浏览器测试**(Ajax**场景使用该方式**)**

请求地址为：http://localhost:9080/springmvc-chapter6/response/ContentType，且把修改请求头Accept改为“Accept=application/json”:



大家可以下载chrome的JSONView插件来以更好看的方式查看json数据，安装地址：https://chrome.google.com/webstore/detail/chklaanhfefbnpoihckbnefhakgolnmc

使用普通客户端测试（服务器之间通信可使用该方式）

```
private static void jsonRequest() throws IOException, URISyntaxException {
    //请求的地址
    String url = "http://localhost:9080/springmvc-chapter6/response/ContentType";
    //①创建Http Request(内部使用HttpURLConnection)
    ClientHttpRequest request =
        new SimpleClientHttpRequestFactory().
            createRequest(new URI(url), HttpMethod.POST);
    //②设置客户端可接受的媒体类型（即需要什么类型的响应体数据）
    request.getHeaders().set("Accept", "application/json");
    //③发送请求并得到响应
    ClientHttpResponse response = request.execute();
    //④得到响应体的编码方式
    Charset charset = response.getHeaders().getContentType().getCharSet();
    //⑤得到响应体的内容
    InputStream is = response.getBody();
    byte bytes[] = new byte[(int)response.getHeaders().getContentLength()];
    is.read(bytes);
    String jsonData = new String(bytes, charset);
    System.out.println("charset : " + charset + ", json data : " + jsonData);
}
```

request.getHeaders().set("Accept", "application/json")：表示客户端只接受（即只消费）json 格式的响应数据；

response.getHeaders()：可以得到响应头，从而可以得到响应体的内容类型和编码、内容长度。

2.2、xml数据

（**1**、服务器端控制器

```
@RequestMapping(value = "/response/ContentType", headers = "Accept=application/xml")
public void response3(HttpServletResponse response) throws IOException {
    //①表示响应的内容区数据的媒体类型为xml格式，且编码为utf-8(客户端应该以utf-8解码)
    response.setContentType("application/xml;charset=utf-8");
    //②写出响应体内容
    String xmlData = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
    xmlData += "<user><username>zhang</username><password>123</password></user>";
    response.getWriter().write(xmlData);
}
```

和生产json数据唯一不同的两点：请求头为“Accept=application/xml”，响应体数据为xml。

（**2**、客户端端接收服务器端**xml**数据响应

使用浏览器测试**(Ajax**场景使用该方式**)**

请求地址为：http://localhost:9080/springmvc-chapter6/response/ContentType，且把修改请求头Accept改为“Accept=application/xml”,和json方式类似，此处不再重复。

使用普通客户端测试（服务器之间通信可使用该方式）

```
    private static void xmlRequest() throws IOException, URISyntaxException {
        //请求的地址
        String url = "http://localhost:9080/springmvc-chapter6/response/ContentType";
        //①创建Http Request(内部使用HttpURLConnection)
        ClientHttpRequest request =
            new SimpleClientHttpRequestFactory().
                createRequest(new URI(url), HttpMethod.POST);
        //②设置客户端可接受的媒体类型（即需要什么类型的响应体数据）
        request.getHeaders().set("Accept", "application/xml");
        //③发送请求并得到响应
        ClientHttpResponse response = request.execute();
        //④得到响应体的编码方式
        Charset charset = response.getHeaders().getContentType().getCharSet();
        //⑤得到响应体的内容
        InputStream is = response.getBody();
        byte bytes[] = new byte[(int)response.getHeaders().getContentLength()];
        is.read(bytes);
        String xmlData = new String(bytes, charset);
        System.out.println("charset : " + charset + ", xml data : " + xmlData);
    }
```

request.getHeaders().set("Accept", "application/xml")：表示客户端只接受（即只消费）xml格式的响应数据；

response.getHeaders()：可以得到响应头，从而可以得到响应体的内容类型和编码、内容长度。

许多开放平台，都提供了同一种数据的多种不同的表现形式，此时我们可以根据Accept请求头告诉它们我们需要什么类型的数据，他们根据我们的Accept来判断需要返回什么类型的数据。

实际项目使用Accept请求头是比较麻烦的，现在大多数开放平台（国内的新浪微博、淘宝、腾讯等开放平台）使用如下两种方式：

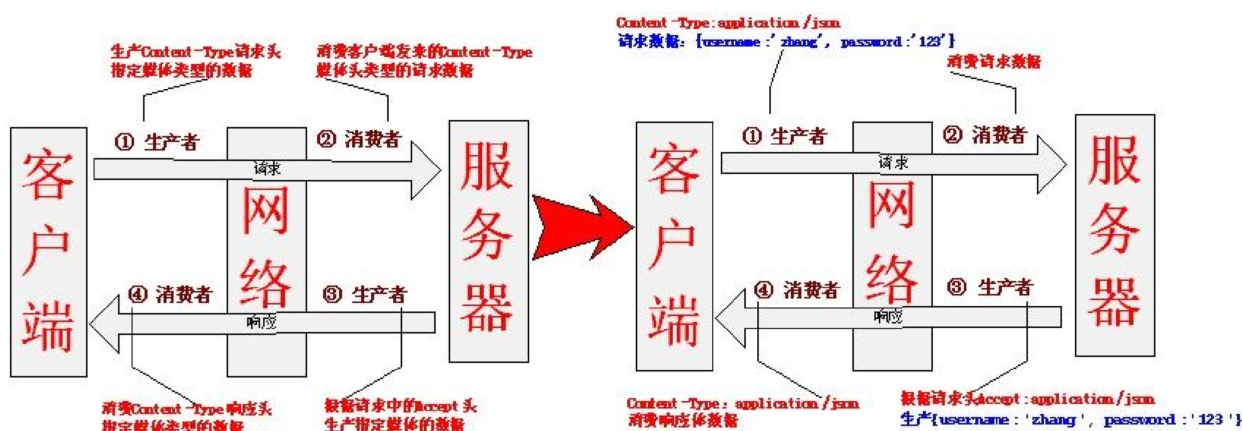
扩展名：如response/ContentType.json response/ContentType.xml方式，使用扩展名表示需要什么类型的数据；

参数：如response/ContentType?format=json response/ContentType?format=xml，使用参数表示需要什么类型的数据；

也就是说，目前我们可以使用如上三种方式实现来告诉服务器我们需要什么类型的数据，但麻烦的是现在有三种实现方式，难道我们为了支持三种类型的数据就要分别进行三种实现吗？当然不要这么麻烦，后续我们会学ContentNegotiatingViewResolver，它能帮助我们做到这一点。

6.6.5.2、生产者消费者流程图

生产者消费者流程，如图**6-10**：



从图6-10可以看出：

请求阶段：客户端是生产者【生产Content-Type媒体类型的请求内容区数据】，服务器是消费者【消费客户端生产的Content-Type媒体类型的请求内容区数据】；

响应阶段：服务器是生产者【生产客户端请求头参数Accept指定的响应体数据】，客户端是消费者【消费服务器根据Accept请求头生产的响应体数据】。

如上生产者/消费者写法无法很好的体现我们分析的生产者/消费者模式，Spring3.1为生产者/消费者模式提供了简化支持，接下来我们学习一下如何在Spring3.1中来实现生产者/消费者模式吧。

6.6.5.3、生产者、消费者限定

Spring3.1开始支持消费者、生产者限定，而且必须使用如下HandlerMapping和HandlerAdapter才支持：

```
<!--Spring3.1开始的注解 HandlerMapping -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping
<!--Spring3.1开始的注解 HandlerAdapter -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter
```

一、功能处理方法是消费者

@RequestMapping(value = "/consumes", consumes = {"application/json"})：此处使用consumes来指定功能处理方法能消费的媒体类型，其通过请求头的“Content-Type”来判断。

此种方式相对使用@RequestMapping的“headers = "Content-Type=application/json"”更能表明你的目的。

服务器控制器代码详解cn.javass.chapter6.web.controller.consumesproduces.ConsumesController；

客户端代码类似于之前的Content-Type中的客户端，详见ConsumesClient.。

二、功能处理方法是生产者

@RequestMapping(value = "/produces", produces = "application/json")：表示将功能处理方法将生产json格式的数据，此时根据请求头中的Accept进行匹配，如请求头“Accept:application/json”时即可匹配;

@RequestMapping(value = "/produces", produces = "application/xml")：表示将功能处理方法将生产xml格式的数据，此时根据请求头中的Accept进行匹配，如请求头“Accept:application/xml”时即可匹配。

此种方式相对使用@RequestMapping的“headers = "Accept=application/json"”更能表明你的目的。

服务器控制器代码详解
cn.javass.chapter6.web.controller.consumesproduces.ProducesController；

客户端代码类似于之前的Content-Type中的客户端，详见ProducesController.。

当你有如下Accept头：

①Accept：text/html,application/xml,application/json

将按照如下顺序进行produces的匹配 ①text/html ②application/xml ③application/json

②Accept：application/xml;q=0.5,application/json;q=0.9,text/html

将按照如下顺序进行produces的匹配 ①text/html ②application/json ③application/xml

q参数为媒体类型的质量因子，越大则优先权越高(从0到1)

③Accept：/,text/*,text/html

将按照如下顺序进行produces的匹配 ①text/html ②text/ ③/*

即匹配规则为：最明确的优先匹配。

代码详见ProducesPrecedenceController1、ProducesPrecedenceController2、ProducesPrecedenceController3。

Accept详细信息，请参考http://tools.ietf.org/html/rfc2616#section-14.1。

三、窄化时是覆盖 而 非继承

如类级别的映射为 @RequestMapping(value="/narrow", produces="text/html")，方法级别的为@RequestMapping(produces="application/xml")，此时方法级别的映射将覆盖类级别的，因此请求头“Accept:application/xml”是成功的，而“text/html”将报406错误码，表示不支持的请求媒体类型。

详见cn.javass.chapter6.web.controller.consumesproduces.NarrowController。

只有生产者/消费者 模式 是 覆盖，其他的使用方法是继承，如headers、params等都是继承。

四、组合使用是“或”的关系

@RequestMapping(produces={"text/html", "application/json"}) ：将匹配“Accept:text/html”或“Accept:application/json”。

五、问题

消费的数据，如JSON数据、XML数据都是由我们读取请求的InputStream并根据需要自己转换为相应的模型数据，比较麻烦；

生产的数据，如JSON数据、XML数据都是由我们自己先把模型数据转换为json/xml等数据，然后输出响应流，也是比较麻烦的。

Spring提供了一组注解（`@RequestBody`、`@ResponseBody`）和一组转换类（`HttpMessageConverter`）来完成我们遇到的问题，详见6.6.8节。

SpringMVC强大的数据绑定（1）——第六章 注解式控制器详解——跟着开涛学SpringMVC

到目前为止，请求已经能交给我们的处理器进行处理了，接下来的事情是要进行收集数据啦，接下来我们看看我们能从请求中收集到哪些数据，如图6-11：



图6-11

1、@RequestParam绑定单个请求参数值；

2、@PathVariable绑定URI模板变量值；

3、@CookieValue绑定Cookie数据值

4、@RequestHeader绑定请求头数据；

5、@ModelValue绑定参数到命令对象；

6、@SessionAttributes绑定命令对象到session；

7、@RequestBody绑定请求的内容区数据并能进行自动类型转换等。

8、@RequestPart绑定“multipart/data”数据，除了能绑定@RequestParam能做到的请求参数外，还能绑定上传的文件等。

除了上边提到的注解，我们还可以通过如HttpServletRequest等API得到请求数据，但推荐使用注解方式，因为使用起来更简单。

接下来先看一下功能处理方法支持的参数类型吧。

6.6.1、功能处理方法支持的参数类型

在继续学习之前，我们需要首先看看功能处理方法支持哪些类型的形式参数，以及他们的具体含义。

一、ServletRequest/HttpServletRequest 和 ServletResponse/HttpServletResponse

```
public String requestOrResponse (
        ServletRequest servletRequest, HttpServletRequest httpServletRequest,
        ServletResponse servletResponse, HttpServletResponse httpServletResponse
    )
```

Spring Web MVC框架会自动帮助我们把相应的Servlet请求/响应（Servlet API）作为参数传递过来。

二、InputStream/OutputStream 和 Reader/Writer

```
public void inputOrOutBody(InputStream requestBodyIn, OutputStream responseBodyOut)
        throws IOException {
responseBodyOut.write("success".getBytes());
}
```

requestBodyIn：获取请求的内容区字节流，等价于request.getInputStream();

responseBodyOut：获取相应的内容区字节流，等价于response.getOutputStream()。

```
public void readerOrWriteBody(Reader reader, Writer writer)
        throws IOException {
    writer.write("hello");
}
```

reader：获取请求的内容区字符流，等价于request.getReader();

writer：获取相应的内容区字符流，等价于response.getWriter()。

InputStream/OutputStream 和 Reader/Writer两组不能同时使用，只能使用其中的一组。

三、WebRequest/NativeWebRequest

WebRequest是Spring Web MVC提供的统一请求访问接口，不仅仅可以访问请求相关数据（如参数区数据、请求头数据，但访问不到Cookie区数据），还可以访问会话和上下文中的数据；NativeWebRequest继承了WebRequest，并提供访问本地Servlet API的方法。

```
public String webRequest(WebRequest webRequest, NativeWebRequest nativeWebRequest) {
    System.out.println(webRequest.getParameter("test"));//①得到请求参数test的值
    webRequest.setAttribute("name", "value", WebRequest.SCOPE_REQUEST);//②
    System.out.println(webRequest.getAttribute("name", WebRequest.SCOPE_REQUEST));
    HttpServletRequest request =
        nativeWebRequest.getNativeRequest(HttpServletRequest.class);//③
    HttpServletResponse response =
        nativeWebRequest.getNativeResponse(HttpServletResponse.class);
        return "success";
    }
```

① webRequest.getParameter：访问请求参数区的数据，可以通过getHeader()访问请求头数据；

② webRequest.setAttribute/getAttribute：到指定的作用范围内取/放属性数据，Servlet定义的三个作用范围分别使用如下常量代表：

：代表请求作用范围；

：代表会话作用范围；

：代表全局会话作用范围，即ServletContext上下文作用范围。

③ nativeWebRequest.getNativeRequest/nativeWebRequest.getNativeResponse：得到本地的Servlet API。

四、**HttpSession**

```
public String session(HttpSession session) {
    System.out.println(session);
    return "success";
}
```

此处的session永远不为null。

注意：session访问不是线程安全的，如果需要线程安全，需要设置AnnotationMethodHandlerAdapter或RequestMappingHandlerAdapter的synchronizeOnSession属性为true，即可线程安全的访问session。

五、命令/表单对象

Spring Web MVC能够自动将请求参数绑定到功能处理方法的命令/表单对象上。

```
@RequestMapping(value = "/commandObject", method = RequestMethod.GET)
public String toCreateUser(HttpServletRequest request, UserModel user) {
    return "customer/create";
}
@RequestMapping(value = "/commandObject", method = RequestMethod.POST)
public String createUser(HttpServletRequest request, UserModel user) {
    System.out.println(user);
    return "success";
}
```

如果提交的表单（包含username和password文本域），将自动将请求参数绑定到命令对象user中去。

六、**Model**、**Map**、**ModelMap**

Spring Web MVC 提供Model、Map或ModelMap让我们能去暴露渲染视图需要的模型数据。

```
@RequestMapping(value = "/model")
public String createUser(Model model, Map model2, ModelMap model3) {
    model.addAttribute("a", "a");
    model2.put("b", "b");
    model3.put("c", "c");
    System.out.println(model == model2);
    System.out.println(model2 == model3);
    return "success";}
```

虽然此处注入的是三个不同的类型（Model model, Map model2, ModelMap model3），但三者是同一个对象，如图6-12所示：

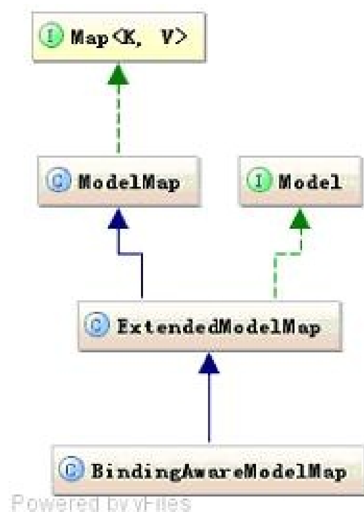


图6-11

AnnotationMethodHandlerAdapter和RequestMappingHandlerAdapter将使用BindingAwareModelMap作为模型对象的实现，即此处我们的形参（Model model, Map model2, ModelMap model3）都是同一个BindingAwareModelMap实例。

此处还有一点需要我们注意：

```
@RequestMapping(value = "/mergeModel")
public ModelAndView mergeModel(Model model) {
    model.addAttribute("a", "a");//①添加模型数据
    ModelAndView mv = new ModelAndView("success");
    mv.addObject("a", "update");//②在视图渲染之前更新③处同名模型数据
    model.addAttribute("a", "new");//③修改①处同名模型数据
    //视图页面的a将显示为"update" 而不是"new"
    return mv;
}
```

从代码中我们可以总结出功能处理方法的返回值中的模型数据（如ModelAndView）会 合并 功能处理方法形式参数中的模型数据（如Model），但如果两者之间有同名的，返回值中的模型数据会覆盖形式参数中的模型数据。

七、**Errors/BindingResult**

```
@RequestMapping(value = "/error1")
public String error1(UserModel user, BindingResult result)
```

```
@RequestMapping(value = "/error2")
public String error2(UserModel user, BindingResult result, Model model) {
```

```
@RequestMapping(value = "/error3")
public String error3(UserModel user, Errors errors)
```

以上代码都能获取错误对象。

Spring3.1之前（使用AnnotationMethodHandlerAdapter）错误对象必须紧跟在命令对象/表单对象之后，如下定义是错误的：

```
@RequestMapping(value = "/error4")
public String error4(UserModel user, Model model, Errors errors)
    }
```

如上代码从Spring3.1开始（使用RequestMappingHandlerAdapter）将能正常工作，但还是推荐“错误对象紧跟在命令对象/表单对象之后”，这样是万无一失的。

Errors及BindingResult的详细使用请参考4.16.2数据验证。

八、其他杂项

```
public String other(Locale locale, Principal principal)
```

java.util.Locale：得到当前请求的本地化信息，默认等价于ServletRequest.getLocale()，如果配置LocaleResolver解析器则由它决定Locale，后续介绍；

java.security.Principal：该主体对象包含了验证通过的用户信息，等价于HttpServletRequest.getUserPrincipal()。

以上测试在cn.javass.chapter6.web.controller.paramtype.MethodParamTypeController中。

其他功能处理方法的形式参数类型（如HttpEntity、UriComponentsBuilder、SessionStatus、RedirectAttributes）将在后续章节详细讲解。

第二部分会介绍注解方式的数据绑定。

SpringMVC强大的数据绑定（2）——第六章 注解式控制器详解——跟着开涛学SpringMVC

6.6.2、@RequestParam绑定单个请求参数值

@RequestParam用于将请求参数区数据映射到功能处理方法的参数上。

```
public String requestparam1(@RequestParam String username)
```

请求中包含username参数（如/requestparam1?username=zhang），则自动传入。

此处要特别注意：右击项目，选择“属性”，打开“属性对话框”，选择“Java Compiler”然后再打开的选项卡将“Add variable attributes to generated class files”取消勾选，意思是不将局部变量信息添加到类文件中，如图6-12所示：

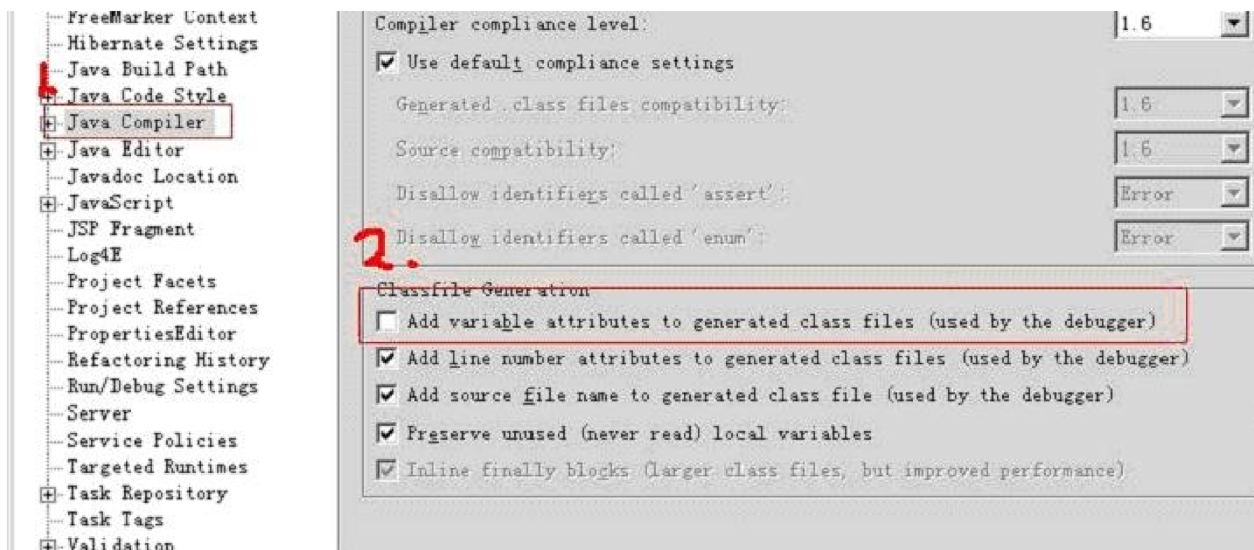


图6-12

当你在浏览器输入URL，如“requestparam1?username=123”时会报如下错误

Name for argument type [java.lang.String] not available, and parameter name information not found in class file either，表示得不到功能处理方法的参数名，此时我们需要如下方法进行入参：

```
public String requestparam2(@RequestParam("username") String username)
```

即通过@RequestParam("username")明确告诉Spring Web MVC使用username进行入参。

接下来我们看一下@RequestParam注解主要有哪些参数：

value：参数名字，即入参的请求参数名字，如username表示请求的参数区中的名字为username的参数的值将传入；

required：是否必须，默认是true，表示请求中一定要有相应的参数，否则将报404错误码；

defaultValue：默认值，表示如果请求中没有同名参数时的默认值，默认值可以是SpEL表达式，如“#{systemProperties['java.vm.version']}”。

```
public String requestparam4(@RequestParam(value="username",required=false) String usernam
```

表示请求中可以没有名字为username的参数，如果没有默认为null，此处需要注意如下几点：

原子类型：必须有值，否则抛出异常，如果允许空值请使用包装类代替。

Boolean包装类型类型：默认Boolean.FALSE，其他引用类型默认为null。

```
public String requestparam5(
@RequestParam(value="username", required=true, defaultValue="zhang") String username)
```

表示如果请求中没有名字为username的参数，默认值为“zhang”。

如果请求中有多个同名的应该如何接收呢？如给用户授权时，可能授予多个权限，首先看下如下代码：

```
public String requestparam7(@RequestParam(value="role") String roleList)
```

如果请求参数类似于url?role=admin&rule=user，则实际roleList参数入参的数据为“admin,user”，即多个数据之间使用“，”分割；我们应该使用如下方式来接收多个请求参数：

```
public String requestparam7(@RequestParam(value="role") String[] roleList)
```

或

```
public String requestparam8(@RequestParam(value="list") List<String> list)
```

到此@RequestParam我们就介绍完了，以上测试代码在cn.javass.chapter6.web.controller.paramtype.RequestParamTypeController中。

6.6.3、@PathVariable绑定URI模板变量值

@PathVariable用于将请求URL中的模板变量映射到功能处理方法的参数上。

```
@RequestMapping(value="/users/{userId}/topics/{topicId}")
public String test(
       @PathVariable(value="userId") int userId,
       @PathVariable(value="topicId") int topicId)
```

如请求的URL为“控制器URL/users/123/topics/456”，则自动将URL中模板变量{userId}和{topicId}绑定到通过@PathVariable注解的同名参数上，即入参后userId=123、topicId=456。代码在PathVariableTypeController中。

6.6.4、@CookieValue绑定Cookie数据值

@CookieValue用于将请求的Cookie数据映射到功能处理方法的参数上。

```
public String test(@CookieValue(value="JSESSIONID", defaultValue="") String sessionId)
```

如上配置将自动将JSESSIONID值入参到sessionId参数上，defaultValue表示Cookie中没有JSESSIONID时默认为空。

```
public String test2(@CookieValue(value="JSESSIONID", defaultValue="") Cookie sessionId)
```

传入参数类型也可以是javax.servlet.http.Cookie类型。

测试代码在CookieValueTypeController中。@CookieValue也拥有和@RequestParam相同的三个参数，含义一样。

6.6.5、@RequestHeader绑定请求头数据

@RequestHeader用于将请求的头信息区数据映射到功能处理方法的参数上。

```
@RequestMapping(value="/header")
public String test(
       @RequestHeader("User-Agent") String userAgent,
       @RequestHeader(value="Accept") String[] accepts)
```

如上配置将自动将请求头“User-Agent”值入参到userAgent参数上，并将“Accept”请求头值入参到accepts参数上。测试代码在HeaderValueTypeController中。

@RequestHeader也拥有和@RequestParam相同的三个参数，含义一样。

6.6.6、@ModelAttribute绑定请求参数到命令对象

@ModelAttribute一个具有如下三个作用：

①绑定请求参数到命令对象：放在功能处理方法的入参上时，用于将多个请求参数绑定到一个命令对象，从而简化绑定流程，而且自动暴露为模型数据用于视图页面展示时使用；

②暴露表单引用对象为模型数据：放在处理器的一般方法（非功能处理方法）上时，是为表单准备要展示的表单引用对象，如注册时需要选择的所在城市等，而且在执行功能处理方法（@RequestMapping注解的方法）之前，自动添加到模型对象中，用于视图页面展示时使用；

③暴露@RequestMapping方法返回值为模型数据：放在功能处理方法的返回值上时，是暴露功能处理方法的返回值为模型数据，用于视图页面展示时使用。

一、绑定请求参数到命令对象

如用户登录，我们需要捕获用户登录的请求参数（用户名、密码）并封装为用户对象，此时我们可以使用@ModelAttribute绑定多个请求参数到我们的命令对象。

```
public String test1(@ModelAttribute("user") UserModel user)
```

和6.6.1一节中的五、命令/表单对象功能一样。只是此处多了一个注解@ModelAttribute("user")，它的作用是将该绑定的命令对象以“user”为名称添加到模型对象中供视图页面展示使用。我们此时可以在视图页面使用${user.username}来获取绑定的命令对象的属性。

绑定请求参数到命令对象支持对象图导航式的绑定，如请求参数包含“?username=zhang&password=123&workInfo.city=bj”自动绑定到user中的workInfo属性的city属性中。

```
@RequestMapping(value="/model2/{username}")
public String test2(@ModelAttribute("model") DataBinderTestModel model) {
```

DataBinderTestModel相关模型请从第三章拷贝过来，请求参数到命令对象的绑定规则详见【4.16.1、数据绑定】一节，URI模板变量也能自动绑定到命令对象中，当你请求的URL中包含“bool=yes&schoolInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&state=blocked”会自动绑定到命令对象上。

当URI模板变量和请求参数同名时，URI模板变量具有高优先权。

二、暴露表单引用对象为模型数据

```
@ModelAttribute("cityList")
public List<String> cityList() {
    return Arrays.asList("北京", "山东");
}
```

如上代码会在执行功能处理方法之前执行，并将其自动添加到模型对象中，在功能处理方法中调用Model 入参的containsAttribute("cityList")将会返回true。

```
@ModelAttribute("user")  //①
public UserModel getUser(@RequestParam(value="username", defaultValue="") String username
//TODO 去数据库根据用户名查找用户对象
UserModel user = new UserModel();
user.setRealname("zhang");
     return user;
}
```

如你要修改用户资料时一般需要根据用户的编号/用户名查找用户来进行编辑，此时可以通过如上代码查找要编辑的用户。

也可以进行一些默认值的处理。

```
@RequestMapping(value="/model1") //②
public String test1(@ModelAttribute("user") UserModel user, Model model)
```

此处我们看到①和②有同名的命令对象，那Spring Web MVC内部如何处理的呢：

(1、首先执行@ModelAttribute注解的方法，准备视图展示时所需要的模型数据；@ModelAttribute注解方法形式参数规则和@RequestMapping规则一样，如可以有@RequestParam等；

（2、执行@RequestMapping注解方法，进行模型绑定时首先查找模型数据中是否含有同名对象，如果有直接使用，如果没有通过反射创建一个，因此②处的user将使用①处返回的命令对象。即②处的user等于①处的user。

三、暴露**@RequestMapping**方法返回值为模型数据

```
public @ModelAttribute("user2") UserModel test3(@ModelAttribute("user2") UserModel user)
```

大家可以看到返回值类型是命令对象类型，而且通过@ModelAttribute("user2")注解，此时会暴露返回值到模型数据（名字为user2）中供视图展示使用。那哪个视图应该展示呢？此时Spring Web MVC会根据RequestToViewNameTranslator进行逻辑视图名的翻译，详见【4.15.5、RequestToViewNameTranslator】一节。

此时又有问题了，@RequestMapping注解方法的入参user暴露到模型数据中的名字也是user2，其实我们能猜到：

（3、@ModelAttribute注解的返回值会覆盖@RequestMapping注解方法中的@ModelAttribute注解的同名命令对象。

四、匿名绑定命令参数

```
public String test4(@ModelAttribute UserModel user, Model model)
或
public String test5(UserModel user, Model model)
```

此时我们没有为命令对象提供暴露到模型数据中的名字，此时的名字是什么呢？Spring Web MVC自动将简单类名（首字母小写）作为名字暴露，如“cn.javass.chapter6.model.UserModel”暴露的名字为“userModel”。

```
public @ModelAttribute List<String> test6()
或
public @ModelAttribute List<UserModel> test7()
```

对于集合类型（Collection接口的实现者们，包括数组），生成的模型对象属性名为“简单类名（首字母小写）”+“List”，如List<String>生成的模型对象属性名为“stringList”，List<UserModel>生成的模型对象属性名为“userModelList”。

其他情况一律都是使用简单类名（首字母小写）作为模型对象属性名，如Map<String, UserModel>类型的模型对象属性名为“map”。

6.6.7、@SessionAttributes绑定命令对象到session

有时候我们需要在多次请求之间保持数据，一般情况需要我们明确的调用HttpSession的API来存取会话数据，如多步骤提交的表单。Spring Web MVC提供了@SessionAttributes进行请求间透明的存取会话数据。

```
//1、在控制器类头上添加@SessionAttributes注解
@SessionAttributes(value = {"user"})    //①
public class SessionAttributeController

//2、@ModelAttribute注解的方法进行表单引用对象的创建
@ModelAttribute("user")    //②
public UserModel initUser()

//3、@RequestMapping注解方法的@ModelAttribute注解的参数进行命令对象的绑定
@RequestMapping("/session1")   //③
public String session1(@ModelAttribute("user") UserModel user)

//4、通过SessionStatus的setComplete()方法清除@SessionAttributes指定的会话数据
@RequestMapping("/session2")   //③
public String session(@ModelAttribute("user") UserModel user, SessionStatus status) {
    if(true) { //④
        status.setComplete();
    }
    return "success";
}
```

@SessionAttributes(value = {"user"})含义：

@SessionAttributes(value = {"user"}) 标识将模型数据中的名字为“user” 的对象存储到会话中（默认HttpSession），此处value指定将模型数据中的哪些数据（名字进行匹配）存储到会话中，此外还有一个types属性表示模型数据中的哪些类型的对象存储到会话范围内，如果同时指定value和types属性则那些名字和类型都匹配的对象才能存储到会话范围内。

包含@SessionAttributes的执行流程如下所示：

① 首先根据@SessionAttributes注解信息查找会话内的对象放入到模型数据中；

② 执行@ModelAttribute注解的方法：如果模型数据中包含同名的数据，则不执行@ModelAttribute注解方法进行准备表单引用数据，而是使用①步骤中的会话数据；如果模型数据中不包含同名的数据，执行@ModelAttribute注解的方法并将返回值添加到模型数据中；

③ 执行@RequestMapping方法，绑定@ModelAttribute注解的参数：查找模型数据中是否有@ModelAttribute注解的同名对象，如果有直接使用，否则通过反射创建一个；并将请求参数绑定到该命令对象；

此处需要注意：如果使用@SessionAttributes注解控制器类之后，③步骤一定是从模型对象中取得同名的命令对象，如果模型数据中不存在将抛出HttpSessionRequiredException Expected session attribute ‘user’(Spring3.1)

或HttpSessionRequiredException Session attribute ‘user’ required - not found in session(Spring3.0)异常。

④ 如果会话可以销毁了，如多步骤提交表单的最后一步，此时可以调用SessionStatus对象的setComplete()标识当前会话的@SessionAttributes指定的数据可以清理了，此时当@RequestMapping功能处理方法执行完毕会进行清理会话数据。

我们通过Spring Web MVC的源代码验证一下吧，此处我们分析的是Spring3.1的RequestMappingHandlerAdapter，读者可以自行验证Spring3.0的AnnotationMethodHandlerAdapter，流程一样：

（1、RequestMappingHandlerAdapter.invokeHandlerMethod

```
//1、RequestMappingHandlerAdapter首先调用ModelFactory的initModel方法准备模型数据：
modelFactory.initModel(webRequest, mavContainer, requestMappingMethod);
//2、调用@RequestMapping注解的功能处理方法
requestMappingMethod.invokeAndHandle(webRequest, mavContainer);
//3、更新/合并模型数据
modelFactory.updateModel(webRequest, mavContainer);
```

（2、ModelFactory.initModel

```
Map<String, ?> attributesInSession = this.sessionAttributesHandler.retrieveAttributes(req
//1.1、将与@SessionAttributes注解相关的会话对象放入模型数据中
mavContainer.mergeAttributes(attributesInSession);
//1.2、调用@ModelAttribute方法添加表单引用对象
invokeModelAttributeMethods(request, mavContainer);
//1.3、验证模型数据中是否包含@SessionAttributes注解相关的会话对象，不包含抛出异常
for (String name : findSessionAttributeArguments(handlerMethod)) {
    if (!mavContainer.containsAttribute(name)) {
        //1.4、此处防止在@ModelAttribute注解方法又添加了会话对象
        //如在@ModelAttribute注解方法调用session.setAttribute("user", new UserModel());
        Object value = this.sessionAttributesHandler.retrieveAttribute(request, name);
        if (value == null) {
            throw new HttpSessionRequiredException("Expected session attribute '" + name
        }
        mavContainer.addAttribute(name, value);
}
```

（3、ModelFactory.invokeModelAttributeMethods

```
for (InvocableHandlerMethod attrMethod : this.attributeMethods) {
    String modelName = attrMethod.getMethodAnnotation(ModelAttribute.class).value();
    //1.2.1、如果模型数据中包含同名数据则不再添加
    if (mavContainer.containsAttribute(modelName)) {
        continue;
    }
    //1.2.2、调用@ModelAttribute注解方法并将返回值添加到模型数据中，此处省略实现代码
}
```

（4、requestMappingMethod.invokeAndHandle 调用功能处理方法，此处省略

（5、ModelFactory.updateMode 更新模型数据

```
//3.1、如果会话被标识为完成，此时从会话中清除@SessionAttributes注解相关的会话对象
if (mavContainer.getSessionStatus().isComplete()){
    this.sessionAttributesHandler.cleanupAttributes(request);
}
//3.2、如果会话没有完成，将模型数据中的@SessionAttributes注解相关的对象添加到会话中
else {
    this.sessionAttributesHandler.storeAttributes(request, mavContainer.getModel());
}
//省略部分代码
```

到此@SessionAtrribute介绍完毕，测试代码在cn.javass.chapter6.web.controller.paramtype.SessionAttributeController中。

另外cn.javass.chapter6.web.controller.paramtype.WizardFormController是一个类似于【4.11、AbstractWizardFormController】中介绍的多步骤表单实现，此处不再贴代码，多步骤提交表单需要考虑会话超时问题，这种方式可能对用户不太友好，我们可以采取隐藏表单（即当前步骤将其他步骤的表单隐藏）或表单数据存数据库（每步骤更新下数据库数据）等方案解决。

6.6.8、@Value绑定SpEL表示式

@Value用于将一个SpEL表达式结果映射到到功能处理方法的参数上。

```
public String test(@Value("#{systemProperties['java.vm.version']}") String jvmVersion)
```

到此数据绑定我们就介绍完了，对于没有介绍的方法参数和注解（包括自定义注解）在后续章节进行介绍。接下来我们学习下数据类型转换吧。

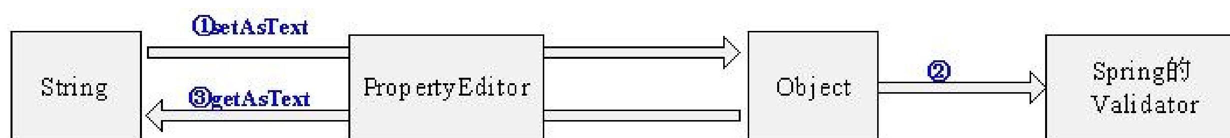
转载请注明出处【http://jinnianshilongnian.iteye.com/blog/1703694】

SpringMVC数据类型转换——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC

7.1、简介

在编写可视化界面项目时，我们通常需要对数据进行类型转换、验证及格式化。

一、在**Spring3**之前，我们使用如下架构进行类型转换、验证及格式化：



流程：

①：类型转换：首先调用PropertyEditor的setAsText（String），内部根据需要调用setValue(Object)方法进行设置转换后的值；

②：数据验证：需要显示调用Spring的Validator接口实现进行数据验证；

③：格式化显示：需要调用PropertyEditor的getText进行格式化显示。

使用如上架构的缺点是：

（1、PropertyEditor被设计为只能String<——>Object之间转换，不能任意对象类型<——>任意类型，如我们常见的Long时间戳到Date类型的转换是办不到的；

（2、PropertyEditor是线程不安全的，也就是有状态的，因此每次使用时都需要创建一个，不可重用；

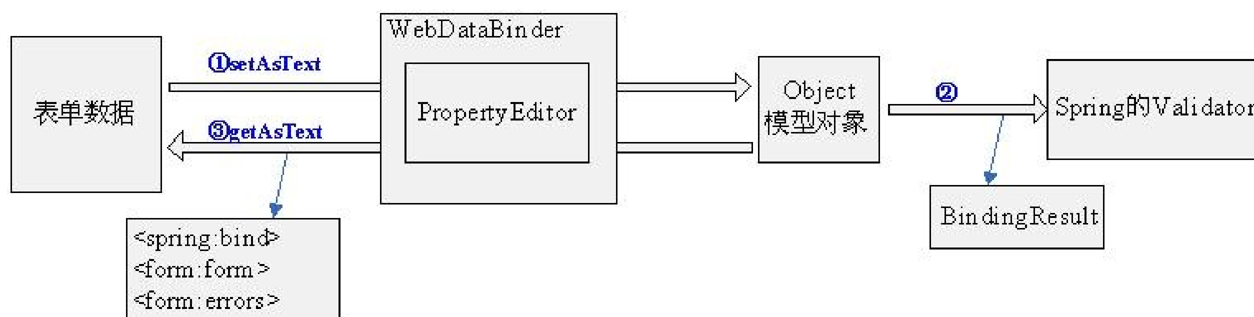
（3、PropertyEditor不是强类型的，setValue（Object）可以接受任意类型，因此需要我们自己判断类型是否兼容；

（4、需要自己编程实现验证，Spring3支持更棒的注解验证支持；

（5、在使用SpEL表达式语言或DataBinder时，只能进行String<--->Object之间的类型转换；

`（6` 、 `不支持细粒度的类型转换/格式化，如UserModel的registerDate需要转换/格式化类似“``2012-05-01``”的`

`**在Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：**`



流程：

```
①、类型转换：首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过PropertyEditor实
```

②：数据验证：在控制器中的功能处理方法中，需要显示的调用Spring的Validator实现并将错误信息添加到BindingResult对象中；

③：格式化显示：在表单页面可以通过如下方式展示通过 `PropertyEditor` 格式化的数据和错误信息：

```
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

首先需要通过如上taglib指令引入spring的两个标签库。

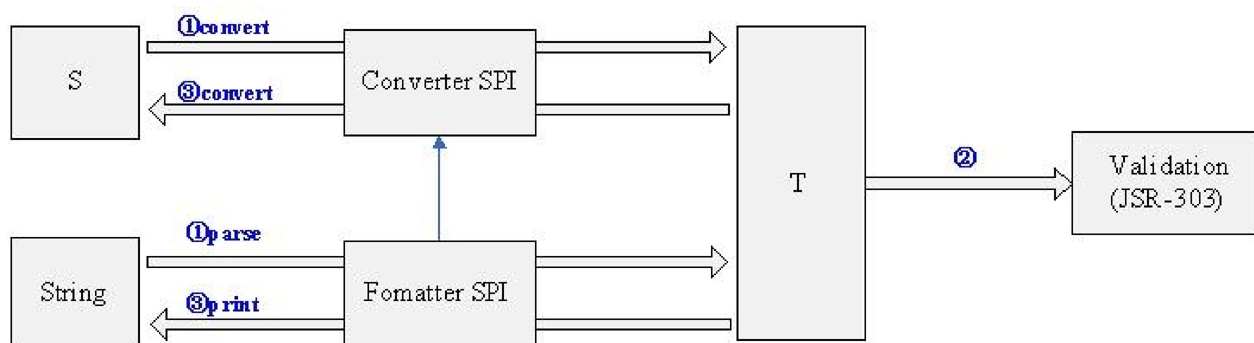
```
//1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）
<spring:bind path="dataBinderTest.phoneNumber">${status.value}</spring:bind>
```

```
//2、通过form标签，内部的表单标签会自动调用命令/表单对象属性对应的PropertyEditor进行格式化显示
<form:form commandName="dataBinderTest">
    <form:input path="phoneNumber"/><!-- 如果出错会显示错误之前的数据而不是空 -->
</form:form>
```

```
//3、显示验证失败后的错误信息
<form:errors></form:errors>
```

如上PropertyEditor和验证API使用起来比较麻烦，而且有许多缺点，因此Spring3提供了更强大的类型转换（Type Conversion）支持，它可以在任意对象之间进行类型转换，不仅仅是String<——>Object；也提供了强大的数据验证支持；同时提供了强大的数据格式化支持。

二、从**Spring3**开始，我们可以使用如下架构进行类型转换、验证及格式化：



流程：

①：类型转换：内部的ConversionService会根据S源类型/T目标类型自动选择相应的Converter SPI进行类型转换，而且是强类型的，能在任意类型数据之间进行转换；

②：数据验证：支持JSR-303验证框架，如将@Valid放在需要验证的目标类型上即可；

③：格式化显示：其实就是任意目标类型---->String的转换，完全可以使用Converter SPI完成。

Spring为了更好的诠释格式化/解析功能提供了Formatter SPI，支持根据Locale信息进行格式化/解析，而且该套SPI可以支持字段/参数级别的细粒度格式化/解析，流程如下：

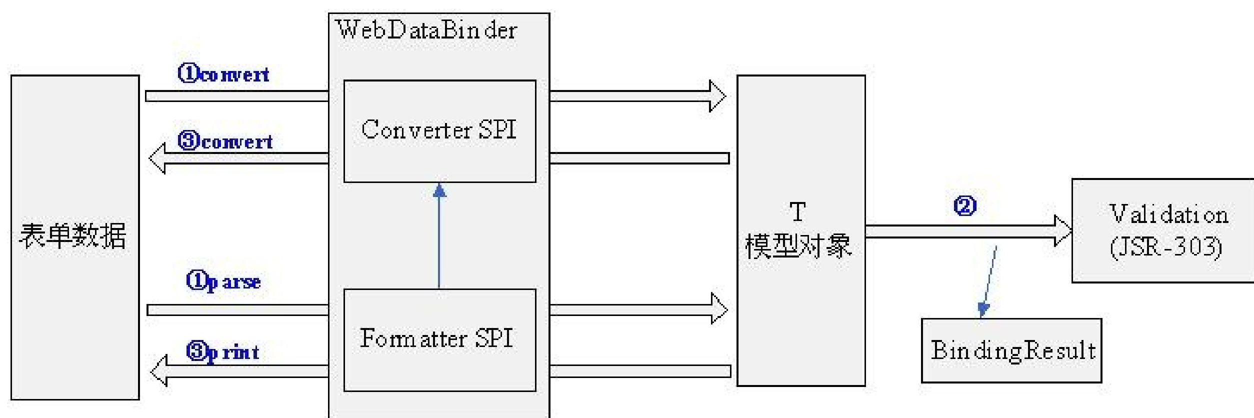
①：类型解析（转换）：String---->T类型目标对象的解析，和PropertyEditor类似；

③：格式化显示：任意目标类型---->String的转换，和PropertyEditor类似。

Formatter SPI最大特点是能进行字段/参数级别的细粒度解析/格式化控制，即使是Converter SPI也是粗粒度的（到某个具体类型，而不是其中的某个字段单独控制），目前Formatter SPI还不是很完善，如果您有好的想法可以到Spring官网提建议。

Formatter SPI内部实现实际委托给Converter SPI进行转换，即约束为解析/格式化String<---->任意目标类型。

在**Spring Web MVC**环境中，数据类型转换、验证及格式化通常是这样使用的：



①、类型转换：首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过Converter SPI实

②：数据验证：使用JSR-303验证框架进行验证；

③：格式化显示：在表单页面可以通过如下方式展示通过`内部通过Converter SPI`格式化的数据和错误信息：

```
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

首先需要通过如上taglib指令引入spring的两个标签库。

```
//1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）
<spring:bind path="dataBinderTest.phoneNumber">${status.value}</spring:bind>
```

```
//2、<spring:eval>标签，自动调用ConversionService并选择相应的Converter SPI进行格式化展示
<spring:eval expression="dataBinderTest.phoneNumber"></spring:eval>
```

如上代码能工作的前提是在RequestMappingHandlerMapping配置了ConversionServiceExposingInterceptor，它的作用是暴露conversionService到请求中以便如<spring:eval>标签使用。

```
//3、通过form标签，内部的表单标签会自动调用命令/表单对象属性对应的PropertyEditor进行格式化显示
<form:form commandName="dataBinderTest">
    <form:input path="phoneNumber"/><!-- 如果出错会显示错误之前的数据而不是空 -->
</form:form>
```

```
//4、显示验证失败后的错误信息
<form:errors></form:errors>
```

接下来我们就详细学习一下这些知识吧。

7.2、数据类型转换

7.2.1、Spring3之前的PropertyEditor

PropertyEditor介绍请参考【4.16.1、数据类型转换】。

一、测试之前我们需要准备好测试环境：

（1、模型对象，和【4.16.1、数据类型转换】使用的一样，需要将DataBinderTestModel模型类及相关类拷贝过来放入cn.javass.chapter7.model包中。

（2、控制器定义：

```
package cn.javass.chapter7.web.controller;
//省略import
@Controller
public class DataBinderTestController {
    @RequestMapping(value = "/dataBind")
    public String test(DataBinderTestModel command) {
            //输出command对象看看是否绑定正确
        System.out.println(command);
        model.addAttribute("dataBinderTest", command);
        return "bind/success";
    }
}
```

(3、Spring配置文件定义，请参考chapter7-servlet.xml，并注册DataBinderTestController：

```
<bean class="cn.javass.chapter7.web.controller.DataBinderTestController"/>
```

（4、测试的URL：

http://localhost:9080/springmvc-chapter7/dataBind?username=zhang&bool=yes&schoolInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&phoneNumber=010-12345678&date=2012-3-18 16:48:48&state=blocked

二、注解式控制器注册**PropertyEditor**：

1、使用**WebDataBinder**进行控制器级别注册**PropertyEditor**（控制器独享）

```
@InitBinder
//此处的参数也可以是ServletRequestDataBinder类型
public void initBinder(WebDataBinder binder) throws Exception {
    //注册自定义的属性编辑器
    //1、日期
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    CustomDateEditor dateEditor = new CustomDateEditor(df, true);
    //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
    binder.registerCustomEditor(Date.class, dateEditor);
    //自定义的电话号码编辑器(和【4.16.1、数据类型转换】一样)
    binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());
}
```

和【4.16.1、数据类型转换】一节类似，只是此处需要通过@InitBinder来注册自定义的PropertyEditor。

2、使用 `**WebBindingInitializer批量注册**` **PropertyEditor**

和【4.16.1、数据类型转换】不太一样，因为我们的注解式控制器是POJO，没有实现任何东西，因此无法注入WebBindingInitializer，此时我们需要把WebBindingInitializer注入到我们的RequestMappingHandlerAdapter或AnnotationMethodHandlerAdapter，这样对于所有的注解式控制器都是共享的。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerA
  <property name="webBindingInitializer">
    <bean class="cn.javass.chapter7.web.controller.support.initializer.MyWebBindingInitia
  </property>
</bean>
```

此时我们注释掉控制器级别通过@InitBinder注册PropertyEditor的方法。

3、全局级别注册PropertyEditor（全局共享）

和【4.16.1、数据类型转换】一节一样，此处不再重复。请参考【4.16.1、数据类型转换】的【全局级别注册PropertyEditor（全局共享）】。

```
接下来我们看一下Spring3提供的更强大的类型转换支持。
```

7.2.2、Spring3开始的类型转换系统

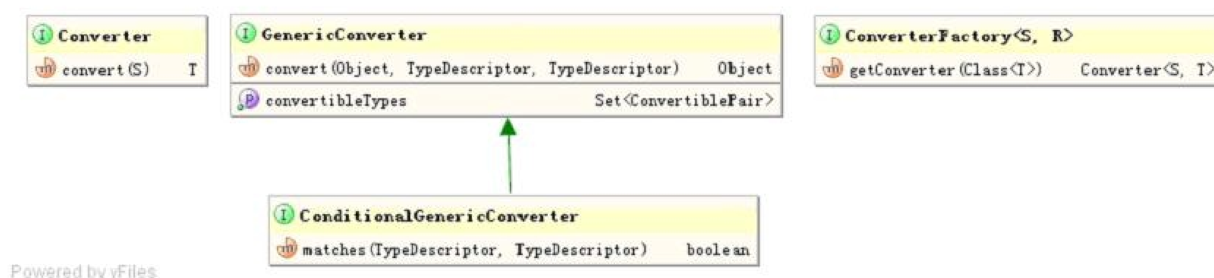
Spring3引入了更加通用的类型转换系统，其定义了SPI接口（Converter等）和相应的运行时执行类型转换的API（ConversionService等），在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来转换外部Bean属性的值到Bean属性需要的类型。

该类型转换系统是Spring通用的，其定义在org.springframework.core.convert包中，不仅仅在Spring Web MVC场景下。目标是完全替换PropertyEditor，提供无状态、强类型且可以在任意类型之间转换的类型转换系统，可以用于任何需要的地方，如SpEL、数据绑定。

Converter SPI完成通用的类型转换逻辑，如java.util.Date<---->java.lang.Long或java.lang.String---->PhoneNumberModel等。

7.2.2.1、架构

1、类型转换器：提供类型转换的实现支持。



一个有如下三种接口：

（**1**、**Converter**：类型转换器，用于转换S类型到T类型，此接口的实现必须是线程安全的且可以被共享。

```
package org.springframework.core.convert.converter;
public interface Converter<S, T> { //① S是源类型 T是目标类型
    T convert(S source); //② 转换S类型的source到T目标类型的转换方法
}
```

示例：请参考cn.javass.chapter7.converter.support.StringToPhoneNumberConverter转换器，用于将String--->PhoneNumberModel。

此处我们可以看到Converter接口实现只能转换一种类型到另一种类型，不能进行多类型转换，如将一个数组转换成集合，如（String[] ----> List<String>、String[]----->List<PhoneNumberModel>等）。

（**2、GenericConverter和ConditionalGenericConverter**：GenericConverter接口实现能在多种类型之间进行转换，ConditionalGenericConverter是有条件的在多种类型之间进行转换。

```
package org.springframework.core.convert.converter;
public interface GenericConverter {
    Set<ConvertiblePair> getConvertibleTypes();
    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

getConvertibleTypes:指定了可以转换的目标类型对；

convert：在sourceType和targetType类型之间进行转换。

```
package org.springframework.core.convert.converter;
public interface ConditionalGenericConverter extends GenericConverter {
    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

matches：用于判断sourceType和targetType类型之间能否进行类型转换。

示例：如org.springframework.core.convert.support.ArrayToCollectionConverter和CollectionToArrayConverter用于在数组和集合间进行转换的ConditionalGenericConverter实现，如在String[]<---->List<String>、String[]<---->List<PhoneNumberModel>等之间进行类型转换。

对于我们大部分用户来说一般不需要自定义GenericConverter, 如果需要可以参考内置的GenericConverter来实现自己的。

（**3、ConverterFactory**：工厂模式的实现，用于选择将一种S源类型转换为R类型的子类型T的转换器的工厂接口。

```
package org.springframework.core.convert.converter;
public interface ConverterFactory<S, R> {
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

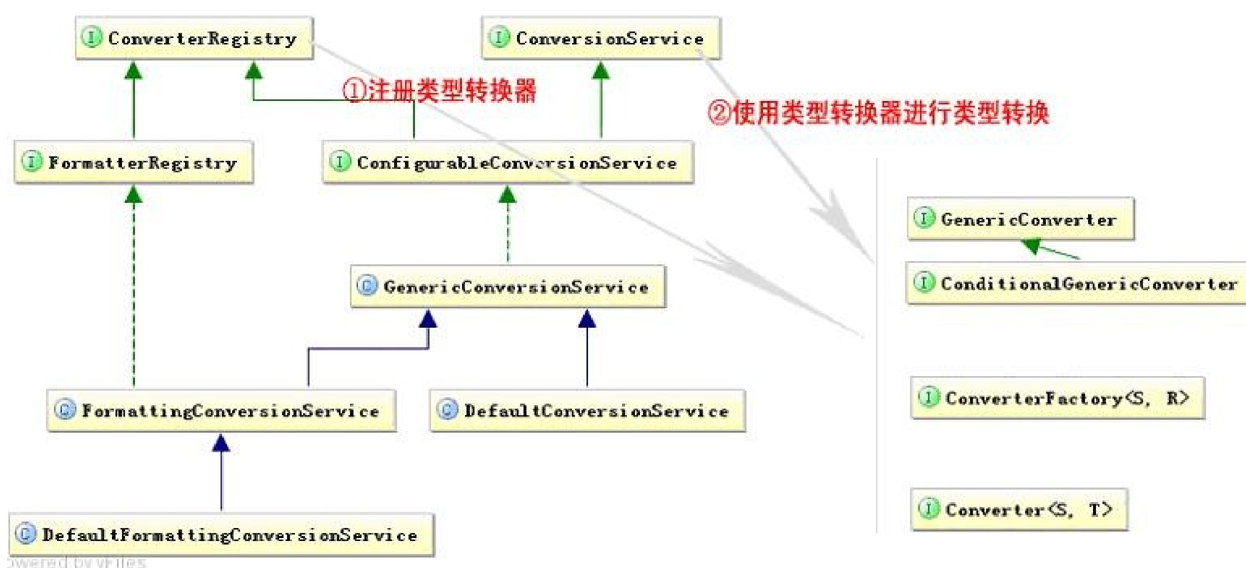
S：源类型；R目标类型的父类型；T：目标类型，且是R类型的子类型；

getConverter：得到目标类型的对应的转换器。

示例：如org.springframework.core.convert.support.NumberToNumberConverterFactory用于在Number类型子类型之间进行转换，如Integer--->Double， Byte---->Integer， Float--->Double等。

对于我们大部分用户来说一般不需要自定义ConverterFactory，如果需要可以参考内置的ConverterFactory来实现自己的。

2、类型转换器注册器、类型转换服务：提供类型转换器注册支持，运行时类型转换API支持。



一共有如下两种接口：

（**1**、**ConverterRegistry**：类型转换器注册支持，可以注册/删除相应的类型转换器。

```
package org.springframework.core.convert.converter;
public interface ConverterRegistry {
    void addConverter(Converter<?, ?> converter);
    void addConverter(Class<?> sourceType, Class<?> targetType, Converter<?, ?> converter
    void addConverter(GenericConverter converter);
    void addConverterFactory(ConverterFactory<?, ?> converterFactory);
    void removeConvertible(Class<?> sourceType, Class<?> targetType);
}
```

可以注册：Converter实现，GenericConverter实现，ConverterFactory实现。

（**2**、**ConversionService**：运行时类型转换服务接口，提供运行期类型转换的支持。

```
package org.springframework.core.convert;
public interface ConversionService {
    boolean canConvert(Class<?> sourceType, Class<?> targetType);
    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);
    <T> T convert(Object source, Class<T> targetType);
    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

convert：将源对象转换为目标类型的目标对象。

Spring提供了两个默认实现（其都实现了ConverterRegistry、ConversionService接口）：

DefaultConversionService:默认的类型转换服务实现；

DefaultFormattingConversionService：带数据格式化支持的类型转换服务实现，一般使用该服务实现即可。

7.2.2.2、Spring内建的类型转换器如下所示：

类名	说明
第一组：标量转换器	
StringToBooleanConverter	String----->Booleantrue:true/on/yes/1；false:false/off/no/0
ObjectToStringConverter	Object----->String调用toString方法转换
StringToNumberConverterFactory	String----->Number（如Integer、Long等）
NumberToNumberConverterFactory	Number子类型(Integer、Long、Double等)<——> Number子类型(Integer、Long、Double等)
StringToCharacterConverter	String----->java.lang.Character取字符串第一个字符
NumberToCharacterConverter	Number子类型(Integer、Long、Double等)——> java.lang.Character
CharacterToNumberFactory	java.lang.Character ——>Number子类型(Integer、Long、Double等)
StringToEnumConverterFactory	String----->enum类型通过Enum.valueOf将字符串转换为需要的enum类型
EnumToStringConverter	enum类型----->String返回enum对象的name()值
StringToLocaleConverter	String----->java.util.Local
PropertiesToStringConverter	java.util.Properties----->String默认通过ISO-8859-1解码
StringToPropertiesConverter	String----->java.util.Properties默认使用ISO-8859-1编码

第二组：集合、数组相关转换器	
ArrayToCollectionConverter	任意S数组---->任意T集合（List、Set）
CollectionToArrayConverter	任意T集合（List、Set）---->任意S数组
ArrayToArrayConverter	任意S数组<---->任意T数组
CollectionToCollectionConverter	任意T集合（List、Set）<---->任意T集合（List、Set）即集合之间的类型转换
MapToMapConverter	Map<---->Map之间的转换
ArrayToStringConverter	任意S数组---->String类型
StringToArrayConverter	String----->数组默认通过“,”分割，且去除字符串的两边空格(trim)
ArrayToObjectConverter	任意S数组---->任意Object的转换(如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换)
ObjectToArrayConverter	Object----->单元素数组
CollectionToStringConverter	任意T集合（List、Set）---->String类型
StringToCollectionConverter	String----->集合（List、Set）默认通过“,”分割，且去除字符串的两边空格(trim)
CollectionToObjectConverter	任意T集合---->任意Object的转换(如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换)
ObjectToCollectionConverter	Object----->单元素集合
第三组：默认（**fallback**）转换器：之前的转换器不能转换时调用	
ObjectToObjectConverter	Object（S）----->Object（T）首先尝试valueOf进行转换、没有则尝试new 构造器(S)
IdToEntityConverter	Id(S)----->Entity(T)查找并调用public static T findEntityName获取目标对象，EntityName是T类型的简单类型
FallbackObjectToStringConverter	Object----->StringConversionService作为恢复使用，即其他转换器不能转换时调用（执行对象的toString()方法）

S：代表源类型，T：代表目标类型

如上的转换器在使用转换服务实现DefaultConversionService和DefaultFormattingConversionService时会自动注册。

7.2.2.3、示例

（1、自定义String----->PhoneNumberModel的转换器

```
package cn.javass.chapter7.web.controller.support.converter;
//省略import
public class StringToPhoneNumberConverter implements Converter<String, PhoneNumberModel>
    Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
    @Override
    public PhoneNumberModel convert(String source) {
        if(!StringUtils.hasLength(source)) {
            //①如果source为空 返回null
            return null;
        }
        Matcher matcher = pattern.matcher(source);
        if(matcher.matches()) {
            //②如果匹配 进行转换
            PhoneNumberModel phoneNumber = new PhoneNumberModel();
            phoneNumber.setAreaCode(matcher.group(1));
            phoneNumber.setPhoneNumber(matcher.group(2));
            return phoneNumber;
        } else {
            //③如果不匹配 转换失败
            throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-1234
        }
    }
}
```

String转换为Date的类型转换器，请参考cn.javass.chapter7.web.controller.support.converter.StringToDateConverter。

(2、测试用例(cn.javass.chapter7.web.controller.support.converter.ConverterTest)

```
@Test
public void testStringToPhoneNumberConvert() {
    DefaultConversionService conversionService = new DefaultConversionService();
    conversionService.addConverter(new StringToPhoneNumberConverter());

    String phoneNumberStr = "010-12345678";
    PhoneNumberModel phoneNumber = conversionService.convert(phoneNumberStr, PhoneNumberM

    Assert.assertEquals("010", phoneNumber.getAreaCode());
}
```

类似于PhoneNumberEditor将字符串“010-12345678”转换为PhoneNumberModel。

```
@Test
public void testOtherConvert() {
    DefaultConversionService conversionService = new DefaultConversionService();

    //"1"--->true（字符串“1”可以转换为布尔值true）
    Assert.assertEquals(Boolean.valueOf(true), conversionService.convert("1", Boolean.cla

    //"1,2,3,4"--->List（转换完毕的集合大小为4）
    Assert.assertEquals(4, conversionService.convert("1,2,3,4", List.class).size());
}
```

其他类型转换器使用也是类似的，此处不再重复。

7.2.2.4、集成到Spring Web MVC环境

（1、注册ConversionService实现和自定义的类型转换器

```
<!-- ①注册ConversionService -->
<bean id="conversionService" class="org.springframework.format.support.
                                                       FormattingConversionServiceFactory
    <property name="converters">
       <list>
            <bean class="cn.javass.chapter7.web.controller.support.
                                   converter.StringToPhoneNumberConverter"/>
            <bean class="cn.javass.chapter7.web.controller.support.
                                   converter.StringToDateConverter">
                <constructor-arg value="yyyy-MM-dd"/>
            </bean>
        </list>
    </property>
</bean>
```

FormattingConversionServiceFactoryBean：是FactoryBean实现，默认使用DefaultFormattingConversionService转换器服务实现；

converters：注册我们自定义的类型转换器，此处注册了String--->PhoneNumberModel和String--->Date的类型转换器。

（2、通过ConfigurableWebBindingInitializer注册ConversionService

```
<!-- ②使用ConfigurableWebBindingInitializer注册conversionService -->
<bean id="webBindingInitializer" class="org.springframework.web.bind.support.
                                                                        ConfigurableWebBi
    <property name="conversionService" ref="conversionService"/>
</bean>
```

此处我们通过ConfigurableWebBindingInitializer绑定初始化器进行ConversionService的注册；

3、注册ConfigurableWebBindingInitializer到RequestMappingHandlerAdapter

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.
                                                            RequestMappingHandlerAdapter"
<property name="webBindingInitializer" ref="webBindingInitializer"/>
</bean>
```

通过如上配置，我们就完成了Spring3.0的类型转换系统与Spring Web MVC的集成。此时可以启动服务器输入之前的URL测试了。

此时可能有人会问，如果我同时使用PropertyEditor和ConversionService，执行顺序是什么呢？内部首先查找PropertyEditor进行类型转换，如果没有找到相应的PropertyEditor再通过ConversionService进行转换。

如上集成过程看起来比较麻烦，后边我们会介绍<mvc:annotation-driven>和@EnableWebMvc，ConversionService会自动注册，后续章节再详细介绍。

SpringMVC数据格式化——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC

7.3、数据格式化

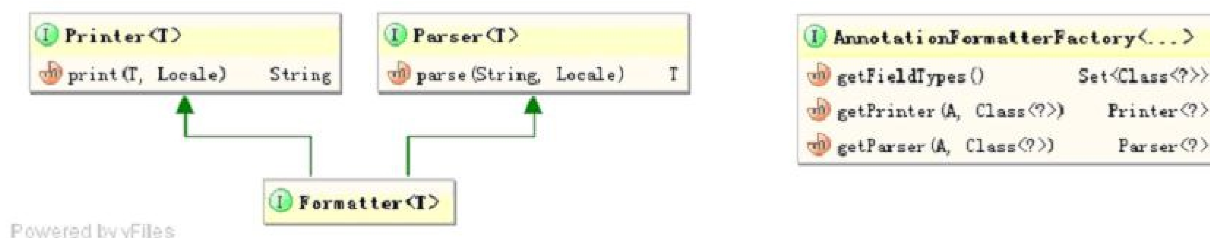
在如Web /客户端项目中，通常需要将数据转换为具有某种格式的字符串进行展示，因此上节我们学习的数据类型转换系统核心作用不是完成这个需求，因此Spring3引入了格式化转换器（Formatter SPI） 和格式化服务API（FormattingConversionService）从而支持这种需求。在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来进行对象的解析和格式化，而且支持细粒度的字段级别的格式化/解析。

Formatter SPI核心是完成解析和格式化转换逻辑，在如Web应用/客户端项目中，需要解析、打印/展示本地化的对象值时使用，如根据Locale信息将java.util.Date---->java.lang.String打印/展示、java.lang.String---->java.util.Date等。

该格式化转换系统是Spring通用的，其定义在org.springframework.format包中，不仅仅在Spring Web MVC场景下。

7.3.1、架构

1、格式化转换器：提供格式化转换的实现支持。



一共有如下两组四个接口：

（**1**、**Printer**接口：格式化显示接口，将T类型的对象根据Locale信息以某种格式进行打印显示（即返回字符串形式）；

```
package org.springframework.format;
public interface Printer<T> {
    String print(T object, Locale locale);
}
```

（**2**、**Parser**接口：解析接口，根据Locale信息解析字符串到T类型的对象；

```
package org.springframework.format;
public interface Parser<T> {
    T parse(String text, Locale locale) throws ParseException;
}
```

解析失败可以抛出java.text.ParseException或IllegalArgumentException异常即可。

（**3**、**Formatter**接口：格式化SPI接口，继承Printer和Parser接口，完成T类型对象的格式化和解析功能；

```
package org.springframework.format;
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

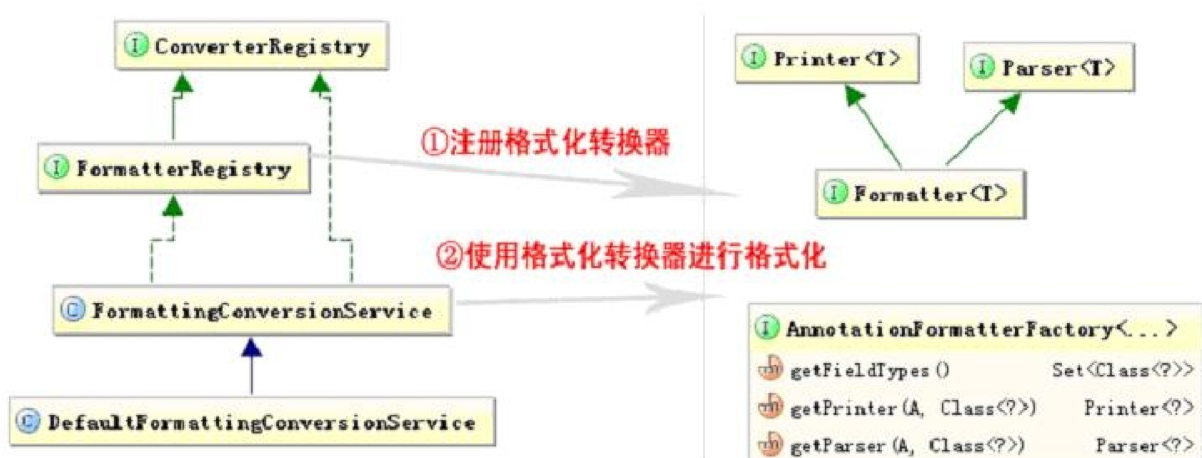
（**4**、**AnnotationFormatterFactory**接口：注解驱动的字段格式化工厂，用于创建带注解的对象字段的Printer和Parser，即用于格式化和解析带注解的对象字段。

```
package org.springframework.format;
public interface AnnotationFormatterFactory<A extends Annotation> {//①可以识别的注解类型
    Set<Class<?>> getFieldTypes();//②可以被A注解类型注解的字段类型集合
    Printer<?> getPrinter(A annotation, Class<?> fieldType);//③根据A注解类型和fieldType类型
    Parser<?> getParser(A annotation, Class<?> fieldType);//④根据A注解类型和fieldType类型获耳

}
```

返回用于格式化和解析被A注解类型注解的字段值的Printer和Parser。如JodaDateTimeFormatAnnotationFormatterFactory可以为带有@DateTimeFormat注解的java.util.Date字段类型创建相应的Printer和Parser进行格式化和解析。

2、格式化转换器注册器、格式化服务：提供类型转换器注册支持，运行时类型转换API支持。



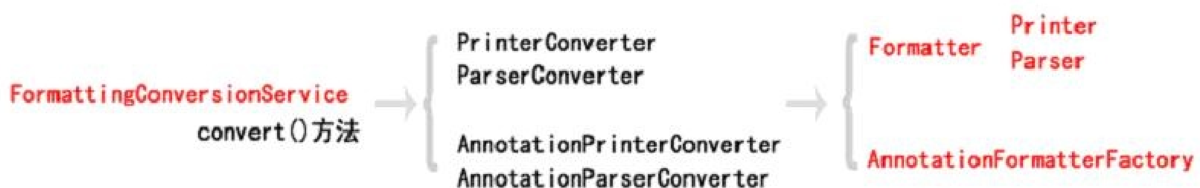
一个有如下两种接口：

（1、FormatterRegistry：格式化转换器注册器，用于注册格式化转换器（Formatter、Printer和Parser、AnnotationFormatterFactory）；

```
package org.springframework.format;
public interface FormatterRegistry extends ConverterRegistry {
    //①添加格式化转换器（Spring3.1 新增API）
    void addFormatter(Formatter<?> formatter);
    //②为指定的字段类型添加格式化转换器
    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);
    //③为指定的字段类型添加Printer和Parser
    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parse
    //④添加注解驱动的字段格式化工厂AnnotationFormatterFactory
    void addFormatterForFieldAnnotation(
                AnnotationFormatterFactory<? extends Annotation> annotationFormatterFacto
}
```

（**2、FormattingConversionService**：继承自ConversionService，运行时类型转换和格式化服务接口，提供运行期类型转换和格式化的支持。

FormattingConversionService内部实现如下图所示：



我们可以看到FormattingConversionService内部实现如上所示，当你调用convert方法时：

(1)若是S类型----->String：调用私有的静态内部类PrinterConverter，其又调用相应的Printer的实现进行格式化；

(2)若是String----->T类型：调用私有的静态内部类ParserConverter，其又调用相应的Parser的实现进行解析；

(3)若是A注解类型注解的S类型----->String：调用私有的静态内部类AnnotationPrinterConverter，其又调用相应的AnnotationFormatterFactory的getPrinter获取Printer的实现进行格式化；

(4)若是String----->A注解类型注解的T类型：调用私有的静态内部类AnnotationParserConverter，其又调用相应的AnnotationFormatterFactory的getParser获取Parser的实现进行解析。

注：S类型表示源类型，T类型表示目标类型，A表示注解类型。

此处可以可以看出之前的Converter SPI完成任意Object与Object之间的类型转换，而Formatter SPI完成任意Object与String之间的类型转换（即格式化和解析，与PropertyEditor类似）。

7.3.2、Spring内建的格式化转换器如下所示：

类名	说明
DateFormatter	java.util.Date<---->String实现日期的格式化/解析
NumberFormatter	java.lang.Number<---->String实现通用样式的格式化/解析
CurrencyFormatter	java.lang.BigDecimal<---->String实现货币样式的格式化/解析
PercentFormatter	java.lang.Number<---->String实现百分数样式的格式化/解析
NumberFormatAnnotationFormatterFactory	@NumberFormat注解类型的数字字段类型<---->String①通过@NumberFormat指定格式化/解析格式②可以格式化/解析的数字类型：Short、Integer、Long、Float、Double、BigDecimal、BigInteger
JodaDateTimeFormatAnnotationFormatterFactory	@DateTimeFormat注解类型的日期字段类型<---->String①通过@DateTimeFormat指定格式化/解析格式②可以格式化/解析的日期类型：joda中的日期类型（org.joda.time包中的）：LocalDate、LocalDateTime、LocalTime、ReadableInstantjava内置的日期类型：Date、Calendar、Longclasspath中必须有Joda-Time类库，否则无法格式化日期类型

NumberFormatAnnotationFormatterFactory和JodaDateTimeFormatAnnotationFormatterFactory（如果classpath提供了Joda-Time类库）在使用格式化服务实现DefaultFormattingConversionService时会自动注册。

7.3.3、示例

在示例之前，我们需要到http://joda-time.sourceforge.net/下载Joda-Time类库，本书使用的是joda-time-2.1版本，将如下jar包添加到classpath：

```
joda-time-2.1.jar
```

7.3.3.1、类型级别的解析/格式化

一、直接使用**Formatter SPI**进行解析/格式化

```
//二、CurrencyFormatter：实现货币样式的格式化/解析
CurrencyFormatter currencyFormatter = new CurrencyFormatter();
currencyFormatter.setFractionDigits(2);//保留小数点后几位
currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）

//1、将带货币符号的字符串“$123.125”转换为BigDecimal("123.00")
Assert.assertEquals(new BigDecimal("123.13"), currencyFormatter.parse("$123.125", Locale.
//2、将BigDecimal("123")格式化为字符串“$123.00”展示
Assert.assertEquals("$123.00", currencyFormatter.print(new BigDecimal("123"), Locale.US))
Assert.assertEquals("￥123.00", currencyFormatter.print(new BigDecimal("123"), Locale.CHI
Assert.assertEquals("￥123.00", currencyFormatter.print(new BigDecimal("123"), Locale.JAP
```

parse方法：将带格式的字符串根据Locale信息解析为相应的BigDecimal类型数据；

print方法：将BigDecimal类型数据根据Locale信息格式化为字符串数据进行展示。

不同于Convert SPI，Formatter SPI可以根据本地化（Locale）信息进行解析/格式化。

其他测试用例请参考cn.javass.chapter7.web.controller.support.formatter.InnerFormatterTest的testNumber测试方法和testDate测试方法。

```
@Test
public void testWithDefaultFormattingConversionService() {
    DefaultFormattingConversionService conversionService = new DefaultFormattingConversio
    //默认不自动注册任何Formatter
    CurrencyFormatter currencyFormatter = new CurrencyFormatter();
    currencyFormatter.setFractionDigits(2);//保留小数点后几位
    currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）
    //注册Formatter SPI实现
    conversionService.addFormatter(currencyFormatter);

    //绑定Locale信息到ThreadLocal
    //FormattingConversionService内部自动获取作为Locale信息，如果不设值默认是 Locale.getDefault
    LocaleContextHolder.setLocale(Locale.US);
    Assert.assertEquals("$1,234.13", conversionService.convert(new BigDecimal("1234.128")
    LocaleContextHolder.setLocale(null);

        LocaleContextHolder.setLocale(Locale.CHINA);
        Assert.assertEquals("￥1,234.13", conversionService.convert(new BigDecimal("1234.
    Assert.assertEquals(new BigDecimal("1234.13"), conversionService.convert("￥1,234.13"
    LocaleContextHolder.setLocale(null);}
```

DefaultFormattingConversionService：带数据格式化功能的类型转换服务实现；

conversionService.addFormatter()：注册Formatter SPI实现；

conversionService.convert(**new** BigDecimal("1234.128"), String.**class**)：用于将BigDecimal类型数据格式化为字符串类型，此处根据“LocaleContextHolder.*setLocale*(locale)”设置的本地化信息进行格式化；

conversionService.convert("￥1,234.13", BigDecimal.**class**)：用于将字符串类型数据解析为BigDecimal类型数据，此处也是根据“LocaleContextHolder.*setLocale*(locale)”设置的本地化信息进行解；

LocaleContextHolder.*setLocale*(locale)：设置本地化信息到ThreadLocal，以便Formatter SPI根据本地化信息进行解析/格式化；

具体测试代码请参考cn.javass.chapter7.web.controller.support.formatter.InnerFormatterTest的testWithDefaultFormattingConversionService测试方法。

三、自定义**Formatter**进行解析/格式化

此处以解析/格式化PhoneNumberModel为例。

（**1**、定义**Formatter SPI**实现

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
public class PhoneNumberFormatter implements Formatter<PhoneNumberModel> {
    Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
    @Override
    public String print(PhoneNumberModel phoneNumber, Locale locale) {//①格式化
        if(phoneNumber == null) {
            return "";
        }
        return new StringBuilder().append(phoneNumber.getAreaCode()).append("-")
                                  .append(phoneNumber.getPhoneNumber()).toString();
    }

    @Override
    public PhoneNumberModel parse(String text, Locale locale) throws ParseException {//②
        if(!StringUtils.hasLength(text)) {
            //①如果source为空 返回null
            return null;
        }
        Matcher matcher = pattern.matcher(text);
        if(matcher.matches()) {
            //②如果匹配 进行转换
            PhoneNumberModel phoneNumber = new PhoneNumberModel();
            phoneNumber.setAreaCode(matcher.group(1));
            phoneNumber.setPhoneNumber(matcher.group(2));
            return phoneNumber;
        } else {
            //③如果不匹配 转换失败
            throw new IllegalArgumentException(String.format("类型转换失败，需要格式[010-1234
        }
    }
}
```

类似于Convert SPI实现，只是此处的相应方法会传入Locale本地化信息，这样可以为不同地区进行解析/格式化数据。

（**2**、测试用例：

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
public class CustomerFormatterTest {
    @Test
    public void test() {
        DefaultFormattingConversionService conversionService = new DefaultFormattingConve
        conversionService.addFormatter(new PhoneNumberFormatter());

        PhoneNumberModel phoneNumber = new PhoneNumberModel("010", "12345678");
        Assert.assertEquals("010-12345678", conversionService.convert(phoneNumber, String

        Assert.assertEquals("010", conversionService.convert("010-12345678", PhoneNumberM
    }
}
```

通过PhoneNumberFormatter可以解析String--->PhoneNumberModel和格式化PhoneNumberModel--->String。

到此，类型级别的解析/格式化我们就介绍完了，从测试用例可以看出类型级别的是对项目中的整个类型实施相同的解析/格式化逻辑。

有的同学可能需要在不同的类的字段实施不同的解析/格式化逻辑，如用户模型类的注册日期字段只需要如“2012-05-02”格式进行解析/格式化即可，而订单模型类的下订单日期字段可能需要如“2012-05-02 20：13：13”格式进行展示。

接下来我们学习一下如何进行字段级别的解析/格式化吧。

7.3.3.2、字段级别的解析/格式化

一、使用内置的注解进行字段级别的解析/格式化：

（1、测试模型类准备：

```
package cn.javass.chapter7.model;
public class FormatterModel {
    @NumberFormat(style=Style.NUMBER, pattern="#,###")
    private int totalCount;
    @NumberFormat(style=Style.PERCENT)
    private double discount;
    @NumberFormat(style=Style.CURRENCY)
    private double sumMoney;

    @DateTimeFormat(iso=ISO.DATE)
    private Date registerDate;

    @DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")
    private Date orderDate;

    //省略getter/setter
}
```

此处我们使用了Spring字段级别解析/格式化的两个内置注解：

@Number：定义数字相关的解析/格式化元数据（通用样式、货币样式、百分数样式），参数如下：

style：用于指定样式类型，包括三种：Style.NUMBER（通用样式） Style.CURRENCY（货币样式） Style.PERCENT（百分数样式），默认Style.NUMBER；

pattern：自定义样式，如patter="#,###"；

@DateTimeFormat：定义日期相关的解析/格式化元数据，参数如下：

pattern：指定解析/格式化字段数据的模式，如"yyyy-MM-dd HH:mm:ss"

iso：指定解析/格式化字段数据的ISO模式，包括四种：ISO.NONE（不使用） ISO.DATE(yyyy-MM-dd) ISO.TIME(hh:mm:ss.SSSZ) ISO.DATE_TIME(yyyy-MM-dd hh:mm:ss.SSSZ)，默认ISO.NONE；

style：指定用于格式化的样式模式，默认“SS”，具体使用请参考Joda-Time类库的org.joda.time.format.DateTimeFormat的forStyle的javadoc；

优先级： pattern 大于 iso 大于 style。

（**2**、测试用例：

```
@Test
public void test() throws SecurityException, NoSuchFieldException {
    //默认自动注册对@NumberFormat和@DateTimeFormat的支持
    DefaultFormattingConversionService conversionService =
                                new DefaultFormattingConversionService();

    //准备测试模型对象
    FormatterModel model = new FormatterModel();
    model.setTotalCount(10000);
    model.setDiscount(0.51);
    model.setSumMoney(10000.13);
    model.setRegisterDate(new Date(2012-1900, 4, 1));
    model.setOrderDate(new Date(2012-1900, 4, 1, 20, 18, 18));

    //获取类型信息
    TypeDescriptor descriptor =
            new TypeDescriptor(FormatterModel.class.getDeclaredField("totalCount"));
    TypeDescriptor stringDescriptor = TypeDescriptor.valueOf(String.class);

    Assert.assertEquals("10,000", conversionService.convert(model.getTotalCount(), descri
    Assert.assertEquals(model.getTotalCount(), conversionService.convert("10,000", string

}
```

TypeDescriptor：拥有类型信息的上下文，用于Spring3类型转换系统获取类型信息的（可以包含类、字段、方法参数、属性信息）；通过TypeDescriptor，我们就可以获取（类、字段、方法参数、属性）的各种信息，如注解类型信息；

conversionService.convert(model.getTotalCount(), descriptor, stringDescriptor)：将totalCount格式化为字符串类型，此处会根据totalCount字段的注解信息（通过descriptor对象获取）来进行格式化；

conversionService.convert("10,000", stringDescriptor, descriptor)：将字符串“10,000”解析为totalCount字段类型，此处会根据totalCount字段的注解信息（通过descriptor对象获取）来进行解析。

（**3**、通过为不同的字段指定不同的注解信息进行字段级别的细粒度数据解析**/**格式化

```
descriptor = new TypeDescriptor(FormatterModel.class.getDeclaredField("registerDate"));
Assert.assertEquals("2012-05-01", conversionService.convert(model.getRegisterDate(), desc
Assert.assertEquals(model.getRegisterDate(), conversionService.convert("2012-05-01", stri

descriptor = new TypeDescriptor(FormatterModel.class.getDeclaredField("orderDate"));
Assert.assertEquals("2012-05-01 20:18:18", conversionService.convert(model.getOrderDate()
Assert.assertEquals(model.getOrderDate(), conversionService.convert("2012-05-01 20:18:18"
```

通过如上测试可以看出，我们可以通过字段注解方式实现细粒度的数据解析/格式化控制，但是必须使用TypeDescriptor来指定类型的上下文信息，即编程实现字段的数据解析/格式化比较麻烦。

其他测试用例请参考cn.javass.chapter7.web.controller.support.formatter.InnerFieldFormatterTest的test测试方法。

二、自定义注解进行字段级别的解析/格式化：

此处以解析/格式化PhoneNumberModel字段为例。

（**1**、定义解析**/**格式化字段的注解类型：

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface PhoneNumber {
}
```

(2、实现**AnnotationFormatterFactory**注解格式化工厂：

```
package cn.javass.chapter7.web.controller.support.formatter;
//省略import
public class PhoneNumberFormatAnnotationFormatterFactory
    implements AnnotationFormatterFactory<PhoneNumber> {//①指定可以解析/格式化的字段注解类型

    private final Set<Class<?>> fieldTypes;
    private final PhoneNumberFormatter formatter;
    public PhoneNumberFormatAnnotationFormatterFactory() {
        Set<Class<?>> set = new HashSet<Class<?>>();
        set.add(PhoneNumberModel.class);
        this.fieldTypes = set;
        this.formatter = new PhoneNumberFormatter();//此处使用之前定义的Formatter实现
    }
    //②指定可以被解析/格式化的字段类型集合
    @Override
    public Set<Class<?>> getFieldTypes() {
        return fieldTypes;
    }
    //③根据注解信息和字段类型获取解析器
    @Override
    public Parser<?> getParser(PhoneNumber annotation, Class<?> fieldType) {
        return formatter;
    }
    //④根据注解信息和字段类型获取格式化器
    @Override
    public Printer<?> getPrinter(PhoneNumber annotation, Class<?> fieldType) {
        return formatter;
    }
}
```

AnnotationFormatterFactory实现会根据注解信息和字段类型获取相应的解析器/格式化器。

（**3**、修改**FormatterModel**添加如下代码：

```
@PhoneNumber
private PhoneNumberModel phoneNumber;
```

（**4**、测试用例

```
 @Test
public void test() throws SecurityException, NoSuchFieldException {
    DefaultFormattingConversionService conversionService =
                                    new DefaultFormattingConversionService();//创建格式
    conversionService.addFormatterForFieldAnnotation(
                    new PhoneNumberFormatAnnotationFormatterFactory());//添加自定义的注解格式

    FormatterModel model = new FormatterModel();
    TypeDescriptor descriptor =
            new TypeDescriptor(FormatterModel.class.getDeclaredField("phoneNumber"));
    TypeDescriptor stringDescriptor = TypeDescriptor.valueOf(String.class);

    PhoneNumberModel value = (PhoneNumberModel) conversionService.convert("010-12345678",
    model.setPhoneNumber(value);

    Assert.assertEquals("010-12345678", conversionService.convert(model.getPhoneNumber(),
}
```

此处使用DefaultFormattingConversionService的addFormatterForFieldAnnotation注册自定义的注解格式化工厂PhoneNumberFormatAnnotationFormatterFactory。

到此，编程进行数据的格式化/解析我们就完成了，使用起来还是比较麻烦，接下来我们将其集成到Spring Web MVC环境中。

7.3.4、集成到Spring Web MVC环境

一、注册**FormattingConversionService**实现和自定义格式化转换器：

```
<bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
  <!—此处省略之前注册的自定义类型转换器-->
  <property name="formatters">
      <list>
           <bean class="cn.javass.chapter7.web.controller.support.formatter.
                                                       PhoneNumberFormatAnnotat
      </list>
  </property>
</bean>
```

其他配置和之前学习7.2.2.4一节一样。

二、示例：

（**1**、模型对象字段的数据解析/格式化：

```
@RequestMapping(value = "/format1")
public String test1(@ModelAttribute("model") FormatterModel formatModel) {
    return "format/success";
}
```

```
totalCount:<spring:bind path="model.totalCount">${status.value}</spring:bind><br/>
discount:<spring:bind path="model.discount">${status.value}</spring:bind><br/>
sumMoney:<spring:bind path="model.sumMoney">${status.value}</spring:bind><br/>
phoneNumber:<spring:bind path="model.phoneNumber">${status.value}</spring:bind><br/>
<!-- 如果没有配置org.springframework.web.servlet.handler.ConversionServiceExposingIntercept
phoneNumber:<spring:eval expression="model.phoneNumber"></spring:eval><br/>

<br/><br/>
<form:form commandName="model">
    <form:input path="phoneNumber"/><br/>
    <form:input path="sumMoney"/>
</form:form>
```

在浏览器输入测试URL：

http://localhost:9080/springmvc-chapter7/format1?totalCount=100000&discount=0.51&sumMoney=100000.128&phoneNumber=010-12345678

数据会正确绑定到我们的formatModel，即请求参数能被正确的解析并绑定到我们的命令对象上，而且在JSP页面也能正确的显示格式化后的数据（即正确的被格式化显示）。

（**2**、功能处理方法参数级别的数据解析：

```
@RequestMapping(value = "/format2")
public String test2(
        @PhoneNumber @RequestParam("phoneNumber") PhoneNumberModel phoneNumber,
        @DateTimeFormat(pattern="yyyy-MM-dd") @RequestParam("date") Date date) {
        System.out.println(phoneNumber);
        System.out.println(date);
        return "format/success2";
}
```

此处我们可以直接在功能处理方法的参数上使用格式化注解类型进行注解，Spring Web MVC 能根据此注解信息对请求参数进行解析并正确的绑定。

在浏览器输入测试URL：

http://localhost:9080/springmvc-chapter7/format2?phoneNumber=010-12345678&date=2012-05-01

数据会正确的绑定到我们的phoneNumber和date上，即请求的参数能被正确的解析并绑定到我们的参数上。

控制器代码位于cn.javass.chapter7.web.controller.DataFormatTestController中。

如果我们请求参数数据不能被正确解析并绑定或输入的数据不合法等该怎么处理呢？接下来的一节我们来学习下绑定失败处理和数据验证相关知识。

SpringMVC数据验证——第七章 注解式控制器的数据验证、类型转换及格式化——跟着开涛学SpringMVC

7.4、数据验证

7.4.1、编程式数据验证

Spring 2.x提供了编程式验证支持，详见【4.16.2 数据验证】章节，在此我们重写【4.16.2.4.1、编程式验证器】一节示例。

（1、验证器实现

复制cn.javass.chapter4.web.controller.support.validator.UserModelValidator

到cn.javass.chapter7.web.controller.support.validator.UserModelValidator。

（2、控制器实现

```
@Controller
public class RegisterSimpleFormController {
    private UserModelValidator validator = new UserModelValidator();

    @ModelAttribute("user")        //① 暴露表单引用对象为模型数据
    public UserModel getUser() {
        return new UserModel();
    }
    @RequestMapping(value = "/validator", method = RequestMethod.GET)
    public String showRegisterForm() {   //② 表单展示
        return "validate/registerAndValidator";
    }
    @RequestMapping(value = "/validator", method = RequestMethod.POST)
    public String submitForm(
            @ModelAttribute("user") UserModel user,
            Errors errors) {           //③ 表单提交
        validator.validate(user, errors);  //1 调用UserModelValidator的validate方法进行验证
        if(errors.hasErrors()) { //2如果有错误再回到表单展示页面
            return showRegisterForm();
        }
        return "redirect:/success";
    }
}
```

在submitForm方法中，我们首先调用之前写的UserModelValidator的validate方法进行验证，当然此处可以直接验证并通过Errors接口来保留错误；此处还通过 Errors接口的hasErrors方法来决定当验证失败时显示的错误页面。

（3、**spring**配置文件**chapter7-servlet.xml**

```
<bean class="cn.javass.chapter7.web.controller.RegisterSimpleFormController"/>
```

（**4**、错误码配置（**messages.properties**），需要执行**NativeToAscii**

直接将【springmvc-chapter4】项目中src下的messages.properties复制到src目录下。

在spring配置文件 `chapter7-servlet.xml中添加` messageSource：

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basename" value="classpath:messages"/>
        <property name="fileEncodings" value="utf-8"/>
        <property name="cacheSeconds" value="120"/>
</bean>
```

（**5**、视图页面（**/WEB-INF/jsp/registerAndValidator.jsp**）

直接将【springmvc-chapter4】项目中的/WEB-INF/jsp/registerAndValidator.jsp复制到当前项目下的/WEB-INF/jsp/validate/registerAndValidator.jsp。

（**6**、启动服务器测试：

在浏览器地址栏输入http://localhost:9080/springmvc-chapter7/validator进行测试，测试步骤和【4.16.2.4.1、编程式验证器】一样。

其他编程式验证的使用，请参考【4.16.2 数据验证】章节。

7.4.2、声明式数据验证

Spring3开始支持JSR-303验证框架，JSR-303支持XML风格的和注解风格的验证，接下来我们首先看一下如何和Spring集成。

7.4.2.1、集成

（**1**、添加**jar**包：

此处使用Hibernate-validator实现（版本：hibernate-validator-4.3.0.Final-dist.zip），将如下jar包添加到classpath（WEB-INF/lib下即可）：

写道dist/lib/required/validation-api-1.0.0.GA.jar JSR-303规范API包 dist/hibernate-validator-4.3.0.Final.jar Hibernate 参考实现

（**2**、在**Spring**配置总添加对**JSR-303**验证框架的支持

```
<!-- 以下 validator  ConversionService 在使用 mvc:annotation-driven 会 自动注册-->
<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <property name="providerClass"  value="org.hibernate.validator.HibernateValidator
        <!-- 如果不加默认到 使用classpath下的 ValidationMessages.properties -->
        <property name="validationMessageSource" ref="messageSource"/>
</bean>
```

此处使用Hibernate validator实现：

validationMessageSource属性：指定国际化错误消息从哪里取，此处使用之前定义的messageSource来获取国际化消息；如果此处不指定该属性，则默认到classpath下的ValidationMessages.properties取国际化错误消息。

通过ConfigurableWebBindingInitializer注册validator：

```
<bean id="webBindingInitializer"
class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
        <property name="conversionService" ref="conversionService"/>
        <property name="validator" ref="validator"/>
</bean>
```

其他配置和之前学习7.2.2.4一节一样。

如上集成过程看起来比较麻烦，后边我们会介绍<mvc:annotation-driven>和@EnableWebMvc，ConversionService会自动注册，后续章节再详细介绍。

（**3**、使用**JSR-303**验证框架注解为模型对象指定验证信息

```
package cn.javass.chapter7.model;
import javax.validation.constraints.NotNull;
public class UserModel {
    @NotNull(message="{username.not.empty}")
    private String username;
}
```

通过@NotNull指定此username字段不允许为空，当验证失败时将从之前指定的messageSource中获取“username.not.empty”对于的错误信息，此处只有通过“{错误消息键值}”格式指定的才能从messageSource获取。

（**4**、控制器

```
package cn.javass.chapter7.web.controller.validate;
//省略import
@Controller
public class HelloWorldController {
    @RequestMapping("/validate/hello")
    public String validate(@Valid @ModelAttribute("user") UserModel user, Errors errors)

        if(errors.hasErrors()) {
            return "validate/error";
        }
        return "redirect:/success";
    }
}
```

通过在命令对象上注解@Valid来告诉Spring MVC此命令对象在绑定完毕后需要进行JSR-303验证，如果验证失败会将错误信息添加到errors错误对象中。

（**5**、验证失败后需要展示的页面（**/WEB-INF/jsp/validate/error.jsp**）

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<form:form commandName="user">
    <form:errors path="*" cssStyle="color:red"></form:errors><br/>
</form:form>
```

（**6**、测试

在浏览器地址栏中输入http://localhost:9080/springmvc-chapter7/validate/hello，即没有username数据，请求后将直接到验证失败界面并显示错误消息"用户名不能为空"，如果请求时带上"?username=zhang"将重定向到成功页面。

到此集成就完成，接下来我们详细学习下有哪些验证约束注解吧。

7.4.2.2、内置的验证约束注解

内置的验证约束注解如下表所示（摘自hibernate validator reference）：

验证注解	验证的数据类型	说明
@AssertFalse	Boolean,boolean	验证注解的元素值是false
@AssertTrue	Boolean,boolean	验证注解的元素值是true
@NotNull	任意类型	验证注解的元素值不是null
@Null	任意类型	验证注解的元素值是null
	BigDecimal，BigInteger, byte,short,	

@Min(value=值)	int, long，等任何Number或CharSequence（存储的是数字）子类型	大于等于@Min指定的value值
@Max（value=值）	和@Min要求一样	验证注解的元素值小于等于@Max指定的value值
@DecimalMin(value=值)	和@Min要求一样	验证注解的元素值大于等于@DecimalMin指定的value值
@DecimalMax(value=值)	和@Min要求一样	验证注解的元素值小于等于@DecimalMax指定的value值
@Digits(integer=整数位数, fraction=小数位数)	和@Min要求一样	验证注解的元素值的整数位数和小数位数上限
@Size(min=下限, max=上限)	字符串、Collection、Map、数组等	验证注解的元素值的在min和max（包含）指定区间之内，如字符长度、集合大小
@Past	java.util.Date,java.util.Calendar;Joda Time类库的日期类型	验证注解的元素值（日期类型）比当前时间早
@Future	与@Past要求一样	验证注解的元素值（日期类型）比当前时间晚
@NotBlank	CharSequence子类型	验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的首位空格
@Length(min=下限, max=上限)	CharSequence子类型	验证注解的元素值长度在min和max区间内
@NotEmpty	CharSequence子类型、Collection、Map、数组	验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0）

@Range(min=最小值, max=最大值)	BigDecimal,BigInteger,CharSequence, byte, short, int, long等原子类型和包装类型	验证注解的元素值在最小值和最大值之间
@Email(regexp=正则表达式,flag=标志的模式)	CharSequence子类型（如String）	验证注解的元素值是Email，也可以通过regexp和flag指定自定义的email格式
@Pattern(regexp=正则表达式,flag=标志的模式)	String，任何CharSequence的子类型	验证注解的元素值与指定的正则表达式匹配
@Valid	任何非原子类型	指定递归验证关联的对象；如用户对象中有个地址对象属性，如果想在验证用户对象时一起验证地址对象的话，在地址对象上加@Valid注解即可级联验证

此处只列出Hibernate Validator提供的大部分验证约束注解，请参考hibernate validator官方文档了解其他验证约束注解和进行自定义的验证约束注解定义。

具体演示实例请参考 cn.javass.chapter7.web.controller.validate.ValidatorAnnotationTestController。

7.4.2.3、错误消息

当验证出错时，我们需要给用户展示错误消息告诉用户出错的原因，因此我们要为验证约束注解指定错误消息。错误消息是通过在验证约束注解的message属性指定。验证约束注解指定错误消息有如下两种方式：

1、硬编码错误消息；

2、从资源消息文件中根据消息键读取错误消息。

一、硬编码错误消息

直接在验证约束注解上指定错误消息，如下所示：

```
@NotNull(message = "用户名不能为空")
@Length(min=5, max=20, message="用户名长度必须在5-20之间")
@Pattern(regexp = "^[a-zA-Z_]\\w{4,19}$", message = "用户名必须以字母下划线开头，可由字母数字下划
private String username;
```

如上所示，错误消息使用硬编码指定，这种方式是不推荐使用的，因为在如下场景是不适用的：

1、在国际化场景下，需要对不同的国家显示不同的错误消息；

2、需要更换错误消息时是比较麻烦的，需要找到相应的类进行更换，并重新编译发布。

二、从资源消息文件中根据消息键读取错误消息

2.1、默认的错误消息文件及默认错误消息键值

默认的错误消息文件是/org/hibernate/validator/ValidationMessages.properties，如下图所示：



默认的错误消息键值如下图所示：

```
javax.validation.constraints.AssertFalse.message = must be false
javax.validation.constraints.AssertTrue.message  = must be true
javax.validation.constraints.DecimalMax.message  = must be less than or equal to {value}
javax.validation.constraints.DecimalMin.message  = must be greater than or equal to {value}
javax.validation.constraints.Digits.message      = numeric value out of bounds (<{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Future.message      = must be in the future
javax.validation.constraints.Max.message         = must be less than or equal to {value}
javax.validation.constraints.Min.message         = must be greater than or equal to {value}
javax.validation.constraints.NotNull.message     = may not be null
javax.validation.constraints.Null.message        = must be null
javax.validation.constraints.Past.message        = must be in the past
javax.validation.constraints.Pattern.message     = must match "{regexp}"
javax.validation.constraints.Size.message        = size must be between {min} and {max}
  省略部分消息键值
```

消息键默认为：验证约束注解的全限定类名.message

在我们之前的测试文件中，错误消息键值是使用默认的，如何自定义错误消息文件和错误消息键值呢？

2.2、自定义的错误消息文件和错误消息键值

自定义的错误消息文件里的错误消息键值将覆盖默认的错误消息文件中的错误消息键值。我们自定义的错误消息文件是具有国际化功能的。

（**1**、定义错误消息文件

在类装载路径的根下创建ValidationMessages.properties文件，如在src目录下创建会自动复制到类装载路径的根下，并添加如下消息键值（需要native2ascii，可以在eclipse里装Properties Editor，自动保存为ASCII码）：

在类装载路径的根下创建ValidationMessages.properties文件，如在src目录下创建会自动复制到类装载路径的根下，并添加如下消息键值（需要native2ascii，可以在eclipse里装Properties Editor，自动保存为ASCII码）：

```
javax.validation.constraints.Pattern.message=用户名必须以字母或下划线开头，后边可以跟字母数字下划线
```

需要在你的spring配置文件WEB-INF/chapter7-servlet.xml修改之前的validator Bean：

```
<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
        <property name="providerClass"
value="org.hibernate.validator.HibernateValidator"/>
</bean>
```

此时错误消息键值的查找会先到classpath下ValidationMessages.properties中找，找不到再到默认的错误消息文件中找。

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/pattern?value=zhan，将看到我们自定义的错误消息显示出来了。

（**2**、使用**Spring**的**MessageSource Bean**进行消息键值的查找

如果我们的环境是与spring集成，还是应该使用Spring提供的消息支持，具体配置如下：

在spring配置文件WEB-INF/chapter7-servlet.xml定义MessageSource Bean：

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basename" value="classpath:messages"/>
        <property name="fileEncodings" value="utf-8"/>
        <property name="cacheSeconds" value="120"/>
</bean>
```

之前我们已经配置过了，在此就不详述了。

在spring配置文件WEB-INF/chapter7-servlet.xml定义的validator Bean，添加如下属性：

```
<property name="validationMessageSource" ref="messageSource"/>
```

验证失败的错误消息键值的查找将使用messageSource Bean进行。

在消息文件src/messages.properties中添加如下错误消息：

```
javax.validation.constraints.Pattern.message=用户名必须以字母或下划线开头，后边可以跟字母数字下划线
```

验证错误注解简单类名.字段名

验证错误注解简单类名.字段类型全限定类名

验证错误注解简单类名

使用的优先级是：从高到低，即最前边的具有最高的优先级，而且以上所有默认的错误消息键优先级高于自定义的错误消息键。

如测试用例cn.javass.chapter7.web.controller.validate.ValidatorAnnotationTestController中的public String pattern(@Valid @ModelAttribute("model") PatternModel model, Errors errors)将自动产生如下错误消息键：

Pattern.model.value=验证错误注解简单类名.验证对象名.字段名

Pattern.value=验证错误注解简单类名.字段名

Pattern.java.lang.String=验证错误注解简单类名.字段类型全限定类名

Pattern=验证错误注解简单类名

（**3**、自定义错误消息键值

之前我们已经学习了硬编码错误消息，及默认的错误消息，在大部分场景下，以上两种方式无法满足我们的需求，因此我们需要自定义错误消息键值。

在验证约束注解上指定错误消息键：

```
package cn.javass.chapter7.web.controller.validate.model;
public class PatternModel {
    @Pattern(regexp = "^[a-zA-Z_][\\w]{4,19}$", message="{user.name.error}")
    private String value;
}
```

我们可以通过验证约束注解的message属性指定错误消息键，格式如"{消息键}"。

在消息文件src/messages.properties中添加如下错误消息：

```
user.name.error=用户名格式不合法
```

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/pattern?value=zhan，将看到我们自定义的错误消息显示出来了。

接下来我们看下如下场景

```
@Length(min=5, max=20, message="{user.name.length.error}")
```

```
user.name.error=用户名长度必须在5-20之间
```

错误消息中的5-20应该是从@Length验证约束注解中获取的，而不是在错误消息中硬编码，因此我们需要占位符的支持：

●如@Length(min=5, max=20, message="{user.name.length.error}")，错误消息可以这样写：用户名长度必须在{min}-{max}之间

错误消息占位符规则：

{验证注解属性名}，如@Length有min和max属性，则在错误消息文件中可以通过{min}和{max}来获取；如@Max有value属性，则在错误消息文件中可以通过{value}来获取。

```
user.name.length.error=用户名长度必须在{min}-{max}之间
```

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/length?value=1，将看到我们自定义的错误消息显示出来了。

7.4.2.4、功能处理方法上多个验证参数的处理

当我们在一个功能处理方法上需要验证多个模型对象时，需要通过如下形式来获取验证结果：

```
@RequestMapping("/validate/multi")
public String multi(
            @Valid @ModelAttribute("a") A a, BindingResult aErrors,
            @Valid @ModelAttribute("b") B b, BindingResult bErrors) {

        if(aErrors.hasErrors()) { //如果a模型对象验证失败
            return "validate/error";
        }
        if(bErrors.hasErrors()) { //如果a模型对象验证失败
            return "validate/error";
        }
        return "redirect:/success";
}
```

每一个模型对象后边都需要跟一个Errors或BindingResult对象来保存验证结果，其方法体内部可以使用这两个验证结果对象来选择出错时跳转的页面。详见cn.javass.chapter7.web.controller.validate.MultiModelController。

在错误页面，需要针对不同的模型来显示错误消息：

```
<form:form commandName="a">
    <form:errors path="*" cssStyle="color:red"></form:errors><br/>
</form:form>
<form:form commandName="b">
    <form:errors path="*" cssStyle="color:red"></form:errors><br/>
</form:form>
```